MASTER EN SCIENCES INFORMATIQUES

Vérification automatique de programmes Prolog basée sur l'interprétation abstraite

Pierard, Benoît

Award date:
1997

Facultés Universitaires
Notre-Dame de la Paix, Namur
Institut d'Informatique

Année académique 1996-1997

Vérification automatique de programmes Prolog basée sur l'interprétation abstraite

Benoit Pierard

Mémoire présenté pour l'obtention du titre de Licencié et Maître en Informatique

Abstract

Ce mémoire décrit le fonctionnement d'un analyseur dédié à la vérification automatique de programmes Prolog, et se basant sur la théorie de l'interprétation abstraite pour atteindre cet objectif.

Il s'agira d'interpréter de manière abstraite un programme Prolog et d'en extraire les propriétés sémantiques utiles pour vérifier sa correction. L'interprétation abstraite se fera à l'aide d'un algorithme générique simulant le fonctionnement de Prolog, et les propriétés concernées sont les modes, les types, le formes, les tailles, les partages de variable et de valeur, et les relations existantes entre les termes d'une substitution. Seront également capturées les propriétés des séquences de substitutions retournées par Prolog. Cette interprétation s'inscrit dans le cadre d'une méthodologie de développement inspirée de Y. Deville et considère de manière prioritaire les spécifications et contraintes définies par l'utilisateur. La vérification d'un programme consistera en la comparaison des propriétés du programme capturées par l'interprétation avec les spécifications et contraintes définies par l'utilisateur. L'analyseur réalisant cette vérification peut être à la base de multiples applications ayant pour objectif l'optimisation de programmes Prolog.

This thesis describes the working of a Prolog programs automated verification analyser based on concepts of abstract interpretation theory.

The idea is to interpret Prolog programs in an abstract way in order to capture its semantics features, and then to verify its correctness. Abstract interpretation will be based on a generic algorithm computing the same semantics as Prolog does. Captured program properties are modes, types, patterns, sizes, possible sharings on terms and relations between terms. Are also captured properties of sequences of substitutions returned by Prolog. Interpretation is part of a development methodology inspired by Y. Deville. An important place is made for users specifications and constraints. Verification consist in comparing computed program features and user's information, the first ones ought to satisfy the last ones. The analyser computing this verification can be used for more dedicated applications about optimising Prolog programs.

Remerciements

Je tiens tout d'abord à remercier Mr Baudouin Le Charlier, professeur à l'Institut et promoteur de ce mémoire, pour sa très grande disponibilité et les nombreuses explications qu'il m'a apportées tout au long de ce travail.

D'autre part, je tiens à remercier tout particulièrement Mr Agostino Cortesi, docteur à l'université de Venise, pour son accueil attentionné et pour toute l'aide précieuse et appréciée qu'il ma apportée lors de mon séjour à Venise.

De même, je remercie toute l'équipe responsable du projet dans lequel s'inscrit ce mémoire. J'y ai découvert un pan de l'informatique très intéressant.

Enfin, je tiens enfin à remercier ma famille pour m'avoir offert cette opportunité unique de réaliser un travail intéressant dans un cadre étranger, ainsi que mes amis qui m'ont soutenu tout au long de ces années, spécialement Eddy, Nathalie, Raphaël, Anne-Françoise et Edouard.

Egalement un tout grand merci à toutes les personnes qui ont collaboré de manière directe ou indirecte à l'élaboration de cet ouvrage. Et tout particulièrement à Bernadette sans qui ce mémoire n'aurai pas connu le déroulement qu'il eut.

Table des matières

1. Introduction

Les langages logiques sont l'implantation de langages déclaratifs basés sur la logique. Ils ont la particularité de séparer les connaissances que l'on a sur les traitements à réaliser et la manière concrète par laquelle ces traitements vont être exécutés. Lorsqu'un programmeur utilise un de ces langages, il devrait en théorie se concentrer uniquement sur la définition des traitements qui constituent un programme, et laisser le modèle d'exécution du langage prendre en charge le moyen de résoudre concrètement ceux-ci. Mais comme nous le verrons, la pratique n'atteint pas cet idéal.

Il existe de multiples langages logiques, dont Prolog (*Programming in Logic*). Prolog donne au programmeur la possibilité de définir des programmes purement logiques. Il bénéficie ainsi de la puissance d'expression d'un langage indépendant de toute implantation. Mais écrire des programmes logiques corrects n'est pas si simple.

Prolog fonctionne sur base d'un modèle d'exécution particulier. Ce modèle doit être suffisamment général pour prendre en charge l'exécution de tous les programmes logiques. Il y parvient en proposant une méthode standard de résolution. Mais parce que cette méthode doit pouvoir faire face à un nombre considérable de cas, elle est souvent mal adaptée à l'exécution d'un programme particulier. En effet, le principal reproche que l'on fait aux programmes Prolog est leur inefficacité, problème auquel il faut apporter des solutions.

Une première solution a été de tirer profit des connaissances que le programmeur a sur un programme particulier mais qu'il ne peut exprimer via les instructions logiques. On lui a alors donné la possibilité d'intervenir dans l'algorithme d'exécution à l'aide d'instructions telles que le *cut* et le *delay*. Cependant, l'introduction de ces instructions dans le langage a fait perdre à celui-ci sa propriété déclarative, ce qui ne fait qu'accroître la difficulté de rédiger des programmes Prolog corrects.

Mais comme c'est par l'usage de ces instructions non logiques qu'il est possible d'obtenir des programmes Prolog optimaux, il est nécessaire de développer une méthodologie de programmation qui permette, dans un premier temps, de spécifier un programme indépendamment de toutes contraintes d'implantation, et dans un second temps, d'intégrer dans ce programme les particularités qui le rendront efficace et qui sont propres au modèle d'exécution de Prolog.

Y. Deville a proposé une telle méthodologie. Cette méthodologie a pour but d'aider le programmeur dans sa tâche en lui permettant de spécifier une procédure de manière purement logique et de préciser les conditions d'application de cette procédure. Il dispose aussi de méthodes de construction et d'optimisation de programmes, connues et vérifiées, qui s'appuieront sur les conditions d'application pour corriger et améliorer ce programme de manière systématique.

Ce sont ces possibilités de vérification et d'optimisation qui nous intéressent. Il est en effet possible de réaliser de manière automatique des analyses de programmes Prolog qui se basent sur les informations données par l'utilisateur pour capturer un maximum de propriétés sur les états dans lesquels le programme pourra se trouver lorsqu'il sera exécuté. Une fois acquises, ces propriétés peuvent servir de base à des opérations de vérification et d'optimisation. D'une part, le programme doit à tout moment se trouver dans les conditions d'utilisation définies par l'utilisateur. Et d'autre part, il est possible de rechercher automatiquement la version exécutable la plus efficace de la procédure qui respecte ces conditions.

Une méthode d'analyse qui peut être appliquée aux programmes Prolog est la méthodologie basée sur l'interprétation abstraite. Cette méthodologie exécute le programme non pas pour calculer des valeurs, mais pour en capturer les propriétés. Les propriétés qu'il est intéressant de dégager d'un programme Prolog et qui peuvent ensuite être utilisées pour vérifier et optimiser ce programme sont les modes, les types, les formes et les tailles des termes ainsi que toutes les relations existantes entre les différents termes et leur taille.

Ce mémoire a pour but de présenter le développement d'un analyseur de programmes Prolog qui vérifie la correction d'un programme sur base des informations fournies par l'utilisateur et calculées par une interprétation abstraite de ce programme. L'analyse va interpréter le programme de manière abstraite afin de capturer les modes, les types, les formes des termes, les partages de variables et de valeurs, et les relations entre les termes et de vérifier s'ils respectent bien les conditions imposées par l'utilisateur. D'autre part, cette vérification ne va pas se baser sur chaque type d'information pris individuellement. Elle va plutôt considérer tous ces types d'information en même temps et utiliser les interactions que l'on peut dégager de ceux-ci pour affiner les résultats de l'analyse. Nous verrons aussi qu'il est possible d'analyser

l'ensemble des résultats obtenus à l'aide d'une nouvelle notion : les séquences abstraites.

La suite de ce mémoire s'organise comme suit. La première partie est un rappel général des notions qui définissent la programmation logique. Il n'est pas nécessaire de la parcourir si l'on connaît déjà cette méthode de programmation. Dans le cas contraire, on y retrouve une définition de toutes les structures utilisées et de toutes les opérations qui permettent de créer et d'exécuter des programmes logiques. On y retrouve également une introduction succincte au langage Prolog[1].

Nous verrons ensuite quelles sont les principes de base qui définissent l'interprétation abstraite, ainsi qu'un exemple montrant que cette méthode d'analyse peut également être utilisée dans d'autres paradigmes de programmation.

Nous adapterons alors l'étude de l'interprétation abstraite au cadre de la programmation Prolog. Cette partie est introduite car elle définit les deux applications principales que l'on peut bâtir sur base de l'interprétation abstraite de programmes Prolog. Ces applications sont la vérification et l'optimisation. Nous verrons quelles sont les propriétés qu'il est utile de capturer pour réaliser ces deux applications, et nous les illustrerons par toute une série d'exemples.

Vient ensuite le coeur de ce mémoire. Il s'agit de la spécification d'un analyseur de programmes Prolog qui capturera un ensemble diversifié de propriétés sur un programme donné et qui mettra ces informations en corrélation afin de dégager des caractéristiques plus fines sur ces programmes. Nous commencerons par présenter l'historique de cet analyseur et les changements qui y ont été apportés. Nous poursuivrons par une vue schématique du fonctionnement de l'algorithme d'interprétation utilisé, et nous exposerons comment nous tenons compte des connaissances de l'utilisateur pour réaliser cette interprétation.

Nous définirons ensuite la spécification complète du domaine d'analyse et des opérations qui captureront les propriétés d'un programme pendant son interprétation. Nous verrons également que le schéma d'analyse retenu est générique : il est défini indépendamment du domaine et peut donc être réutilisé pour d'autres sortes d'analyses. Il suffit pour cela de redéfinir le domaine et les opérations qui capturent l'information.

Nous décrirons ensuite l'état actuel de l'analyseur, et le niveau d'implantation auquel nous sommes arrivés. Nous donnerons également la trace complète de l'analyse d'une procédure récursive afin d'illustrer de manière très concrète comment l'interprétation se déroule et comment les propriétés sont capturées à chacune de ces étapes.

---

[1] Nous supposons que ce langage est connu du lecteur.

2. La programmation logique

La programmation logique est un des multiples paradigmes de programmation existants. Elle vise une classe de programmes pour lesquels, comme nous le verrons plus loin, il n'est pas nécessaire de définir comment un problème va être solutionné, mais plutôt quel est le problème à solutionner. Dans ce cadre, la spécification du problème *est* le programme. Celui-ci *n'est pas* la spécification de la solution.

Comme ce mémoire a pour but de présenter une méthode de vérification de programmes logiques, il est utile de rappeler toutes les notions qui constituent ce paradigme et qui seront manipulées tout au long de ce travail. Ce qui suit ne consiste pas une présentation de toutes les notions existantes, mais uniquement de celles qui ont un intérêt pour la présentation des chapitres suivants. Les définitions données sont principalement extraites de [2] et [8], auxquelles on peut se rapporter pour plus d'informations sur la théorie de la programmation logique.

Ce chapitre commence par montrer que la programmation logique est basée sur la logique mathématique et donne un ensemble de définitions nécessaires. Nous présentons ensuite les deux notions les plus importantes en programmation logique : la substitution et l'opération d'unification. Ces deux notions permettent en effet d'exécuter un programme logique[1] et nous verrons les opérations qui constituent cette exécution.

[1] Nous verrons de manière intuitive « comment » on exécute un programme logique.

Enfin, nous verrons un langage de programmation logique bien spécifique, Prolog, ainsi qu'un bref aperçu de son fonctionnement.

2.1 LA LOGIQUE MATHÉMATIQUE

La programmation logique est basée sur la logique mathématique. Toute logique mathématique est constituée de trois composants : une syntaxe, une sémantique opérationnelle et une sémantique déclarative. La syntaxe constitue l'ensemble des règles à respecter lors de l'écriture de formules logiques. Elle définit les objets manipulés par cette logique et les interactions possibles entre eux. La sémantique opérationnelle définit de manière abstraite la dérivation de nouvelles formules valides à partir d'un ensemble de formules de base. La sémantique déclarative s'attache à calculer la valeur des formules écrites par rapport à un domaine donné, et donne ainsi un « sens » au programme.

En logique mathématique, la manipulation des variables et des valeurs est particulière. On ne manipule pas des assignations de valeurs à des variables mais des implications universelles : une fois qu'une variable est instanciée avec une valeur, la valeur de cette variable ne change plus.

Il existe différentes sortes de logiques mathématiques : la logique des propositions, la logique des prédicats du premier ordre et la logique des prédicats d'ordre supérieur. Pour chacune d'elle, il faut définir les trois composants cités plus haut. Seule la logique des prédicats du premier ordre nous intéresse.

2.2 LA LOGIQUE DES PRÉDICATS DU PREMIER ORDRE

La logique des prédicats de premier ordre est une logique mathématique. Pour cette raison, il est nécessaire de définir la syntaxe et les sémantiques déclarative et procédurale qui la composent. Ces trois composants sont décrits ci-dessous.

2.2.1 La syntaxe

La syntaxe de la logique des prédicats du premier ordre définit toutes les formules de premier ordre bien formées que l'on peut écrire. Les formules de premier ordre sont composées de variables quantifiées existentiellement ou universellement et de formules atomiques liées par des connecteurs logiques et construites au moyen de fonctions.

Par exemple,
$\forall X\ \forall Y\ (grand\text{-}parent(X,Y)) \Leftrightarrow (\exists Z\ (parent(X,Z) \wedge parent(Z,Y)))$
est une formule de premier ordre bien formée et exprime le fait que tout X est grand parent de Y si et seulement si il existe un Z quelconque parent de Y et dont le parent est X.

Plus formellement, on a l'ensemble de définitions suivant.

- Un **symbole de variable** est une chaîne de caractères finie composée de chiffres, de lettres et de « _ » et commençant par une majuscule ou « _ ».

- Un **symbole de fonction** ou de **prédicat** est une chaîne de caractères finie et composée de chiffres, de lettres et de « _ » et commençant par une minuscule ou un chiffre. Un tel symbole est souvent accompagné du nombre d'arguments de la fonction ou du prédicat (l'arité).

- Une **constante** est une fonction d'arité 0.

- Un **terme** est défini inductivement par les règles suivantes :

  i) les constantes sont des termes ;
  ii) les variables sont des termes ;
  iii) Si $t_1,...t_n$ sont des termes et si f/n est une fonction d'arité n ($n > 0$), alors $f(t_1,...,t_n)$ est un terme.

- Un **terme clos** est un terme qui ne contient pas de variables.

- Si p/n est un prédicat d'arité n et si $t_1,...,t_n$ sont des termes, alors $p(t_1,...,t_n)$ est un **atome**.

- Un **atome clos** est un atome qui ne contient pas de variables.

- Un **prédicat** est un atome retournant une valeur de vérité. Une **fonction** est un atome retournant un terme.

- Le **constructeur de liste** « . » d'arité 2 est un symbole de fonction.

- Le symbole de **liste vide** « [] » est un symbole de constante.

- Les **symboles arithmétiques** « + », « - », etc. sont des symboles de prédicats et ont leur arité habituelle.

- Les **connecteurs logiques** sont $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$.

- Les **quantificateurs** sont $\exists$ et $\forall$.

- Une **formule bien formée (wff)** est définie inductivement par :

  i) un atome est une wff ;
  ii) Si F et G sont des wffs, alors $\neg$F, F $\wedge$ G, F $\vee$ G, F $\Rightarrow$ G, F $\Leftarrow$ G et F $\Leftrightarrow$ G sont des wffs ;
  iii) Si X est une variable et F une wff, alors $\forall$ X(F) et $\exists$ X(F) sont des wffs.

- Un **littéral** est un atome ou la négation d'un atome.

- Une **expression** est un littéral, un terme, une conjonction de littéraux ou une disjonction de littéraux.

- L'ensemble des wffs définit le **langage de premier-ordre**.

On a aussi que :

- La **portée** de $\forall$ X(F) et $\exists$ X(F) est F. Une **occurrence** de X est dite « **libre** » si elle n'est pas dans la portée d'un quantificateur. Sinon, elle est dite « **liée** ».

- Une wff est « **fermée** » si elle ne contient aucune occurrence libre de variable.

Si une wwf contient des occurrences libres de variables, n'importe quelle valeur peut être affectée à ces variables sans modifier la signification de la wff. Dans ce cas, un quantificateur universel peut être ajouté à la wff de manière telle que sa portée couvre une variable libre particulière. Si on ajoute un tel quantificateur à toutes les variables libres d'une wff, on obtient une wff équivalente à la première mais sans occurrence libre de variables. Donc, *toute wff peut être fermée par ajout de quantificateurs universels portant sur les variables libres.*

Ce langage est trop riche pour être automatisé efficacement. Pour faciliter cette automatisation, on introduit les notions suivantes.

- Une **clause** est une wff de la forme

  $\forall \mathbf{X_1} ... \forall \mathbf{X_n} (\mathbf{L_1} \vee ... \vee \mathbf{L_m})$

  où L_i est un littéral ($i \in \{1,...,m\}$) et X_j est une variable présente dans L_i ($j \in \{1,...,n\}$). Les variables d'une clause sont implicitement quantifiées universellement en tête de la clause. Comme toute wff peut être fermée par ajout de quantificateurs universels portant sur les variables libres, *toute wff fermée peut être exprimée sous forme de clause.*

- Une **clause de Horn** est une clause où il y a 0 ou 1 atome non nié.

  Le fait qu'une clause de Horn ne puisse contenir qu'un seul élément non nié peut rendre impossible la représentation de certaines clauses sous forme de clauses de Horn. Par exemple, si on a la clause $a \Rightarrow (b \vee c)$, qui est équivalent à la clause $\neg a \vee b \vee c$, on a que b et c sont des atomes non niés. Et donc la clause n'est pas une clause de Horn. Ce sont les clauses de Horn qui sont retenues pour développer une programmation logique simple et efficace.

2.2.2 La sémantique opérationnelle

La sémantique opérationnelle définit la manière de transformer des wffs en d'autres wffs. On dira que, si une formule F peut être dérivée d'un ensemble de formules de base S, F est une formule valide. On note cette propriété S |- F.

La dérivation de formules se fait sur base d'axiomes (formules admises sans démonstration) et utilise les règles de dérivation suivantes : Modus Ponens, Modus Tollens, le chaînage, la généralisation et la spécialisation. Par ces dérivations, on établit P |- Q si l'on parvient à obtenir la clause vide à partir d'un programme P et d'une question Q posée au programme. Divers algorithmes ont été créés pour dériver des formules à partir de P et Q de manière à atteindre au plus vite le but fixé, c'est à dire,

obtenir une clause vide. Le principe de résolution de Robinson-Herbrand est un de ces algorithmes et s'applique sur des wffs sous forme de clauses.

2.2.3 La sémantique déclarative

La sémantique déclarative donne la signification des formules en termes de valeurs de vérité, de modèles et de conséquences logiques. Les valeurs de vérité sont « vrai » et « faux ». Un modèle est basé sur l'interprétation que l'on donne à un langage de premier ordre. Une interprétation est une assignation de valeurs issues d'un domaine choisi aux différents éléments composant le langage. Si la valeur de vérité d'une formule est « vrai » dans une interprétation particulière, on dit que cette interprétation est un modèle pour la formule. Enfin, on dit que la formule fermée F est une conséquence logique d'un ensemble de formules fermées S si tous les modèles de S sont des modèles de F. On note alors S|=F.

L'exécution d'un programme consiste à vérifier si une formule F (la question) est une conséquence logique du programme S. En d'autres mots, on recherche un ensemble de valeurs qui, lorsqu'elles sont assignées à la question, et dans le cadre d'un programme donné et d'une interprétation donnée, donnent la valeur « vrai » à cette question.

2.2.4 Relation entre la sémantique déclarative et la sémantique opérationnelle

Dans le cas des clauses de Horn, on a démontré les relations de correction (partielle) et de complétude entre ces deux sémantiques. La correction exprime le fait que toutes les valeurs calculées pendant l'exécution d'un programme et qui sont assignées à la question pour laquelle elles ont été calculées, font que cette question est une conséquence logique du programme. De plus, chacune d'elles est la plus générale possible : si l'on remplace une de ces valeurs par une valeur plus générale, la question n'est plus une conséquence logique du programme.

La complétude établit le résultat inverse. Elle établit que tout ensemble de valeurs moins générales que les valeurs calculées par le programme donnent la valeur de vérité « vrai » à la question pour laquelle elles ont été calculées.

Ces deux résultats impliquent une équivalence[2] entre les deux sémantiques. Si on établit qu'une formule fermée F peut être dérivée d'un ensemble de formules fermées S, on a établit que la formule F est une conséquence logique de S. L'inverse est aussi vrai. Or, la sémantique déclarative correspond à la définition d'un problème : le programmeur, lorsqu'il rédige ses formules, décrit le problème qu'il doit solutionner. Le problème, *et rien que lui*, constitue le programme. Le programmeur n'a donc plus à considérer des détails d'implantation propres au langage qu'il utilise. Par contre, ce langage prend en charge toute la sémantique opérationnelle et s'occupe de tous les aspects propres à l'exécution. C'est la raison pour laquelle la programmation logique est une programmation déclarative.

---

[2] Cette équivalence ne sera pas démontrée ici.

2.3 LA PROGRAMMATION LOGIQUE BASÉE SUR LA LOGIQUE DES PRÉDICATS DU PREMIER ORDRE SOUS FORME DE CLAUSES DE HORN

Des séquences de formules sont rédigées dans le langage logique, et constituent des programmes. Lorsqu'on pose une question au programme, on « exécute le programme », c'est à dire qu'on recherche des valeurs telles que P |= Q. La recherche de telles valeurs fait appel à deux notions importantes : les substitutions et l'unification. Les substitutions représentent l'ensemble des valeurs trouvées à un moment donné. L'unification est le processus qui permet de trouver ces valeurs.

2.3.1 Les substitutions

Les substitutions peuvent être représentées par des ensembles d'associations, par des fonctions ou par des systèmes d'équations. On ne retiendra que la représentation par ensemble d'associations.

Une **substitution** est un ensemble fini d'associations X_i/t_i de la forme $\{X_1/t_1,...,X_m/t_m\}$ où X_i est une variable, t_i est un terme, $X_i \neq X_j$ $(i \neq j)$ et $X_i \neq t_i$ $(i, j \in \{1,...,m\})$. Une association X_i/t_i est une instanciation de X_i, c'est à dire qu'on a assigné la valeur t_i à X_i. Une substitution peut être vide. Elle est alors notée ε.

Le **domaine d'une substitution** α est l'ensemble $\{X_1,...,X_m\}$, et est noté $dom(\alpha)$. Le **codomaine** de α est $\{t_1,...,t_m\}$ et est noté $cod(\alpha)$. L'ensemble des variables appartenant à $cod(\alpha)$ est noté $varcod(\alpha)$.

La **restriction d'une substitution** $\theta = \{X_1/t_1,...,X_m/t_m\}$ à l'ensemble des variables S, noté $\theta_{|S}$, est la substitution obtenue à partir de θ dans laquelle on a supprimé toutes les associations X_i/t_i telles que $X_i \notin S$.

L'application d'une substitution $\theta = \{X_1/t_1,...,X_m/t_m\}$ à une expression E, notée $E\theta$, dénote l'expression obtenue à partir de E en remplaçant dans E toute occurrence de X_i par le terme t_i $(1 \leq i \leq n)$. Après cette opération, on dit que X_i est instancié par t_i, ou encore que t_i est une instance de X_i.

Les propriétés suivantes sont équivalentes :

i) $\sigma = \tau$ (σ et τ sont deux substitutions);
ii) $\forall$ expression E, $E\sigma = E\tau$;
iii) $\forall$ variable $X \in dom(\sigma) \cup dom(\tau)$, $X\sigma = X\tau$.

La **composition des substitutions** $\sigma = \{X_1/t_1,...X_m/t_m\}$ et $\tau = \{Y_1/u_1,...,Y_n/u_n\}$, notée $\sigma\circ\tau$ ou $\sigma\tau$, est la subtitution obtenue à partir de $\{X_1/t_1\tau$ $,...,X_m/t_m\tau, Y_1/u_1,...,Y_n/u_n\}$ où on supprime toutes les associations $X_i/t_i\tau$ telles que $t_i\tau = X_i$ et toutes les associations Y_j/u_j telles que $Y_j \in dom(\sigma)$ $(1 \leq i \leq m, 1 \leq j \leq n)$.

Propriétés de la composition de substitutions[3] :

Soient θ, σ et τ trois substitutions, alors :

i) $\theta\varepsilon = \theta = \varepsilon\theta$;
ii) $\forall$ expression E, $(E\sigma)\tau = E(\sigma\tau)$;
iii) $(\theta\sigma)\tau = \theta(\sigma\tau)$.

Une **substitution σ est plus générale qu'une substitution τ** $\Leftrightarrow$ $\exists$ substitution γ:$\tau=\sigma\gamma$. On notera cette relation $\sigma\leq\tau$. Cette relation $\leq$ est partielle, réflexive et transitive. Elle définit un pré-ordre partiel sur les substitutions.

Une **substitution σ est équivalente à une substitution τ** si et seulement si $\sigma\leq\tau \wedge \tau\leq\sigma$. On notreta cette relations $\sigma \equiv \tau$.

Une **substitution de renomination** est une substitution de la forme $\{X_1/Y_1,...,X_n/Y_n\}$ où les X_i et Y_i sont des variables distinctes.

2.3.2 L'unification

L'unification est un algorithme visant à créer des associations X_i/t_i. Lors de l'unification, les atomes et les termes ont le même comportement. A cause de cela, on généralise ces deux notions à celle de terme étendu.

Un **terme étendu** (ou e-terme) est un terme ou un atome.

Une **substitution θ unifie deux e-termes** E et F $\Leftrightarrow$ les instances $E\theta$ et $F\theta$ sont syntaxiquement identiques. Dans ce cas, E et F sont **unifiables** et θ est **l'unificateur** de E et F. L'unificateur θ est **l'unificateur le plus général** (mgu) de E et F s'il est plus général que tous les autres unificateurs de E et F.

Ces notions peuvent être étendues aux systèmes d'équations entre e-termes. Un système d'équations est sous forme résolue si et seulement s'il a la forme $\{X_1=t_1,...,X_n=t_n\}$ où les X_i sont des variables distinctes qui n'apparaîssent dans aucun des t_i. Dans ce cas, la substitution $\{X_1/t_1,...,X_n/t_n\}$ est appelée la substitution associée au système d'équations.

L'unification dans la logique du premier ordre est décidable. Pour cela, plusieurs algorithmes réalisant l'unification de deux termes existent. L'un d'entre eux travaille à partir de deux e-termes E et F, et consiste à résoudre l'équation E = F. Plus généralement, il permet de résoudre une série d'équations $E_i=F_i$.

Algorithme d'unification de deux termes.

On cherche à unifier les e-termes E et F. Soit S le système d'équations {E=F}, il faut répèter la liste des opérations suivantes le plus grand nombre de fois.

---

[3] Une démonstration ce l'équivalence de ces propriétés se trouve dans [8].

Algorithme d'unification de deux termes.

On cherche à unifier les e-termes E et F. Soit S le système d'équations {E=F}, il faut répèter la liste des opérations suivantes le plus grand nombre de fois.

Etape	*Précondition pour la réalisation de l'opération d'unification*	*Opération d'unfication*
1)	S est vide	Stop avec Succès
2)	S n'est pas vide	Choisir une équation Eq présente dans S ;
2.1)	Eq : $f(t_1,...,t_n) = g(u_1,...,u_m)$ où f syntaxiquement différent de g	Stop avec Echec
2.2)	Eq : $f(t_1,...,t_n) = f(u_1,...,u_m)$ où m, $n \geq 0$, $m \neq n$	Stop avec Echec
2.3)	Eq : $f(_{t1},...,t_n) = f(u_1,...,u_n)$ où $n \geq 0$	Remplacer Eq par les équations $t_1=u_1,...,t_n=u_n$
2.4)	Eq : X = X avec X variable	Retirer Eq de S
2.5)	Eq : t = X avec X variable et t terme non variable	Remplacer Eq par l'équation X = t
2.6)	Eq : X = t avec X variable distincte du terme t, X apparaît dans S / {Eq}, X n'apparaît pas dans t	Remplacer X par t dans toutes les équations sauf Eq
2.7)	Eq : X = t avec X variable distincte du terme t, X apparaît dans S / {Eq}, X apparaît dans t	Stop avec Echec
3)	Retourner en 1)	

Interprétation des résultats de cet algorithme :

Les termes E et F sont unifiables si et seulement si l'algorithme appliqué au système d'équations {E=F} s'arrête avec succès et retourne un système d'équations sous forme résolue. Dans ce cas, la substitution associée à ce système résolu constitue le mgu de E et F.

Exemple d'application de l'algorithme.

Soient E = f(g(X), h(a,Z)) et F = f(g(g(Z)),h(a,Y)).
Le système S initial est égal à {f(g(X),h(a,Z))=f(g(g(Z)),h(a,Y))}.
Après avoir appliqué l'algorithme, on obtient le système {X=g(Z), a=a, Z=Y}
qui, une fois résolu, donne {X=g(Y), Z=Y}.
La substitution associée à ce système résolu est {X/g(Y), Z/Y} et constitue le mgu de E et F.

2.3.3 L'exécution d'un programme

Une question posée à un programme a la forme d'une conjonction d'atomes que l'on écrit $\leftarrow G_1,...,G_m$. L'exécution du programme consiste en la transformation successive de cette conjonction en d'autres conjonctions d'atomes. Une transformation est appelée une étape de déduction et consiste en :

1. On choisit arbitrairement un atome G_i et une clause du programme $H \leftarrow B_1,...,B_n$ ($n \geq 0$) dont la tête H s'unifie avec G_i.

2. On renomme les variables de la clause.

3. On calcule le mgu de G_i et H, c'est à dire qu'on applique l'algorithme d'unification sur G_i et H. Cet algorithme nous indique si les deux e-termes s'unifient et, si oui, nous retourne leur mgu, soit θ.

4. Si l'unification est un succès, on calcule $\leftarrow(G_1,...G_{i-1},B_1,...,B_n,G_{i+1},...,G_m)\theta$. Sinon, stop avec échec.

La répétition de ces transformations se termine[4] quand la conjonction d'atomes est vide. Dans ce cas, on dit qu'elle est un succès. Cette répétition est appelée SLD-derivation ou SLD-refutation ou SLD-resolution.

La composition des θ_i produits pendant la SLD-resolution et restreinte aux variables présentes dans la question consitue une substitution de réponse calculée (computed answer substitution, ou CAS) pour la question.

2.4 PROLOG

Cette partie est consacrée à une présentation très succincte du langage logique Prolog. Une définition beaucoup plus détaillée de ce langage peut être trouvée dans [3] et [4]. Bien que le sujet de ce mémoire traite de l'analyse de programmes Prolog, nous n'insistons pas trop sur sa définition car, comme nous dans les chapitres consacrés à l'interprétation abstraite de programmes Prolog et à la présentation de l'analyseur, nous ne tenons pas compte du fonctionnement de Prolog, nous nous intéressons seulement aux résultats qu'il retourne. D'autre part, une description du langage Prolog n'est pas nécessaire ici car nous réutiliserons des outils informatiques qui nous donnent directement une représentation interne d'un programme Prolog.

2.4.1 Introduction

Prolog est un langage qui permet de rédiger des programmes logiques composés de faits et d'implications et d'exécuter ces programmes à l'aide des substitutions et de l'opération d'unification. Les implications y sont représentées sous forme de prédicats et les faits sont considérés comme vrai dans une interprétation donnée. Une fois ces prédicats et ces faits représentés, il est possible de poser des questions les concernant. Prolog se charge de trouver toutes les réponses possibles à ces questions. Une question prend la forme d'une implication sans conséquent, et donc la forme d'un prédicat. Répondre à la question, c'est évaluer le prédicat. L'évaluation d'un prédicat consiste à trouver des valeurs pour les variables présentes dans le prédicat qui rendent celui-ci vrai. Ces valeurs doivent être telles que le prédicat soit vérifié. Avant l'évaluation, les variables n'ont aucune valeur. Après l'évaluation, les variables ont des

[4] Elle peut ne pas se terminer.

valeurs connues. Si aucune variable n'apparaît dans le prédicat avant son évaluation, celle-ci revient à tester la présence d'un fait dans l'ensemble des faits représentés dans le programme ou de vérifier si le fait peut être déduit du programme.

Une variable prend donc une valeur lors de l'évaluation. Une fois qu'une valeur est donnée à une variable, toutes les occurrences de cette variable ont la même valeur. Les variables sont locales aux prédicats dans lesquels elles apparaissent. Chaque fois que le prédicat est utilisé lors de l'évaluation, on a une nouvelle instance de ses variables. Si on utilise plusieurs fois le même prédicat, on renomme ses variables autant de fois, et chaque fois différemment, afin d'en assurer l'unicité.

En programmation logique, il n'y a pas de différence entre les données en entrée et celles en sortie. Un argument « en entrée » est simplement un argument dont on connaît la valeur avant l'évaluation, et cette valeur ne change pas. Un argument « en sortie » est un argument dont on ignore la valeur avant l'évaluation, celle-ci lui étant donnée pendant l'évaluation. Il n'y a donc a priori pas de paramètres « en entrée » à donner pour obtenir des paramètres « en sortie ». Il y a simplement « des paramètres ». L'évaluation du prédicat ne cherche pas à calculer des valeurs de sortie à partir de données en entrée. Elle cherche à trouver des valeurs pour les paramètres telles que le prédicat soit vérifié. Et s'il existe plusieurs valeurs telles que le prédicat est vérifié, Prolog les recherchera toutes.

Ainsi, un même prédicat peut être utilisé de différentes manières. Si tous les paramètres ont des valeurs connues avant l'évaluation, celle-ci revient à tester si ces valeurs respectent la relation définie. Prolog informe alors l'utilisateur si oui ou non les valeurs données sont satisfaisantes. Si certains des paramètres, voire tous, n'ont pas de valeurs connues au moment de l'appel, l'évaluation recherche les valeurs « manquantes » qui, combinées aux valeurs connues, vérifient le prédicat.

Par exemple, le prédicat « plus(X,Y,Z) » dit que « la somme de X et Y donne Z ». L'évaluation de « plus(1,2,3) » teste si « la somme de 1 et 2 donne 3 ». Par contre l'évaluation de « plus(1,5,Z) » recherche la valeur de Z telle que 1+5 donne Z. Prolog ne trouvera qu'une seule valeur satisfaisant cette relation. Dans ce cas, on utilise le prédicat dans un but bien précis : effectuer un calcul. On peut aussi évaluer « plus(1,Z,5) » et effectuer ainsi la soustraction de 5 et 1. Enfin, il est possible d'évaluer « plus(X,Y,Z) ». Prolog va alors trouver toutes les additions possibles. On effectue ici une recherche.

L'évaluation est non déterministe : Prolog calcule toutes les combinaisons de valeurs possibles satisfaisant le prédicat.

2.4.2 Structure d'un programme Prolog

Un programme Prolog est constitué d'une séquence de prédicats. Chaque prédicat est représenté par une clause de Horn. L'ensemble des clauses de même nom forment une procédure. Chaque clause représente une implication à zéro ou un conséquent. Elle est composée d'une partie gauche (la tête) et d'une partie droite (le corps). La tête est composée d'un seul atome, qui correspond à la conséquence de l'implication. Le corps est composé d'une séquence d'atomes. Cette séquence peut être

vide. Exécuter une tête de clause, c'est vérifier tous les atomes de cette clause. Exécuter une procédure, c'est exécuter les différentes clauses de la procédure et retourner toutes les valeurs calculées.

2.4.3 Exemple

La procédure select/3 définit une relation entre les termes X, L et LS. Cette relation est vérifiée si L et LS sont des listes, si X est un élément de L et si LS est la liste obtenue à partir de L où l'on a retiré une occurrence de X.

select(X,L,LS) :- L = [H|T], H=X, LS=T, list(T).
select(X,L,LS) :- L = [H|T], LS=[H|TS], select(X,T,TS).

Cet exemple est important car il nous servira tout au long du mémoire.

Quelques résultats de cette procédure sont, par exemple :
- *l'appel* select(1,[1,2,3],LS) *retourne* L=[2,3] ;
- *l'appel* select(1,[1,2,3,1,2],LS) *retourne* L=[2,3,1,2] et L=[1,2,3,2] ;
- *l'appel* select(X,[1,2,3,1], LS) *retourne* X=1,LS=[2,3,1] *et* X=1,LS=[1,2,3] *et* X=2,LS=[1,3,1] *et* X=3,LS=[1,2,1] ;
- *l'appel* select ([a,b],L,[a,b]) *retourne* L=[[a,b],a,b].

3. La méthode de programmation de Yves Deville

Le développement de programmes Prolog[1] corrects ne se fait pas sans difficultés. Parce qu'il existe certaines différences entre la sémantique déclarative d'un langage Prolog et sa sémantique procédurale, la simple traduction d'un programme correct du point de vue logique en une version utilisable en Prolog ne suffit pas à assurer que cette version se comportera de la manière attendue. Le perfectionnement des langages logiques visant à rapprocher ces deux sémantiques constitue une première solution, mais elle est insuffisante à elle seule. Comme nous le verrons à l'aide d'un exemple tiré de [7], le comportement d'une procédure Prolog est lui-même difficile à prédire de façon sûre. Le temps consacré à sa compréhension peut devenir (beaucoup) trop important. Et le développement intensif de programmes logiques en est freiné, d'autant plus si les programmeurs travaillent en groupe. Pour cette raison, « il est aussi important de présenter des méthodologies de construction de programmes logiques visant un large usage de langages logiques. »[7]. En d'autres mots, il est utile de définir un cadre de travail qui non seulement supporte le développement des applications, mais qui aide aussi les personnes concernées à comprendre un programme et à s'assurer que celui-ci est correct.

[1] Ou de tout autre langage logique.

Pour ces raisons, Y. Deville a élaboré une méthodologie de programmation qui permet de passer d'une spécification de procédure à une implantation en Prolog respectant cette spécification. Les méthodes de dérivation utilisées pour passer d'une spécification de procédure à son implantation sont connues et vérifiées. On peut ainsi se concentrer sur la recherche d'une représentation logique d'une procédure problème plutôt que de rechercher le moyen de faire fonctionner le programme.

La suite de ce chapitre constitue une introduction à cette méthodologie. Elle s'inspire très fortement de [7] et présente les notions les plus importantes qui seront utilisées dans la suite de ce mémoire. Nous commencerons, comme nous l'avons dit, par donner un exemple illustrant les difficultés rencontrées lors du développement de programmes logiques. Ensuite, nous verrons quels sont les concepts et méthodes proposés par Y. Deville pour un développement systématique de programmes logiques corrects. Enfin, nous verrons ce que nous en retiendrons, et ce que nous y ajoutons, pour réaliser la vérification automatique de programmes développée dans ce mémoire.

3.1 Problèmes rencontrés pendant le développement de programmes logiques

On montre ici quels sont les problèmes liés aux programmes logiques sur base de la procédure efface/3. Cet exemple est extrait de [7]. La spécification de cette procédure est : « la procédure efface(X,L,LS) crée la liste LS sur base de la liste L à laquelle on retire la première occurrence de X. Si X n'est pas présent dans L, la procédure échoue. »

Trois programmes Prolog peuvent, à première vue, solutionner ce problème :

1. efface(X,L,LS) :- L = [H|T], LS = T.
   efface(X,L,LS) :- L=[H|T], LS=[H|TS], efface(X,T,TS).

2. efface(X,L,LS) :- L = [H|T], LS = T, !.
   efface(X,L,LS) :- L=[H|T], LS=[H|TS], efface(X,T,TS).

3. efface(X,L,LS) :- L = [H|T], LS = T.
   efface(X,L,LS) :- L=[H|T], H≠X, LS=[H|TS], efface(X,T,TS).

La première version de ce programme est la plus simple : on décompose la liste L jusqu'à la première occurrence de X, et on construit LS à partir de la queue de la liste et par ajouts successifs en tête de LS des éléments qui précédent X dans L. La seconde version est une optimisation de la première : lorsqu'une première solution a été retournée, on sait qu'il n'y a pas d'autres listes dont la première occurrence de X est différente de celle qui vient d'être traitée. L'instruction « cut » (!) permet d'interrompre la recherche une fois la première solution trouvée. Enfin, la troisième version contient une condition de vérification (H≠X) qui assure que la seconde clause de la procédure n'est utilisée que quand le premier terme de L est différent de X. D'autres versions de cette procédure existent, notamment celles qui regroupent les propriétés des deuxième et

troisième versions présentées ici, mais ces exemples suffisent pour montrer la difficulté d'analyser le comportement d'une procédure logique.

L'exécution de chacune de ces versions doit donner le même résultat. C'est ce qu'on va tester en réalisant une série d'exécutions de chaque procédure à partir des mêmes données. Le tableau suivant montre les résultats obtenus suite à chacune de ces exécutions. Il est composé de 13 tests pour lesquels on définit les paramètres en entrée utilisés, ainsi que le résultat attendu sur base de la spécification. Les trois dernières colonnes nous donnent les valeurs réellement calculées par chacune des versions. Ces résultats sont 'yes' ou 'no' lorsque tous les paramètres ont une valeur initiale connue : 'yes' nous indique que la relation est vérifiée avec ces paramètres, 'no' nous affirme le contraire. Dans les autres cas, les résultats attribuent une valeur à chaque variable utilisée comme paramètre, et ces valeurs vérifient la relation.

Tous les résultats des procédures qui apparaissent barrés sont faux. Les autres correspondent au comportement attendu de la procédure.

Test n° :	X initial	L initial	LS initial	Résultats attendus	Résultats procédure 1	Résultats procédure 2	Résultats procédure 3
1	2	[1,2,3,2,4]	[1,3,2,4]	yes	yes	yes	yes
2	2	[1,2,3,2,4]	[1,2,3,4]	no	~~yes~~	~~yes~~	no
3	2	[2|3]	3	?	~~yes~~	~~yes~~	~~yes~~
4	2	[1,2,3,2,4]	LS	LS=[1,3,2,4]	LS=[1,3,2,4] ~~LS=[1,2,3,4]~~	LS=[1,3,2,4]	LS=[1,3,2,4]
5	0	[1,2,3,2,4]	LS	no	no	no	no
6	X	[1,2,3,2,4]	[1,3,2,4]	X=2	X=2	X=2	~~no~~
7	X	[1,2,3,2,4]	[1,2,3,4]	no	~~X=2~~	~~X=2~~	no
8	2	L	[1,3,2,4]	L=[2,1,3,2,4] L=[1,2,3,2,4] L=[1,3,2,2,4]	L=[2,1,3,2,4] L=[1,2,3,2,4] L=[1,3,2,2,4] ~~L=[1,3,2,4,2]~~	~~L=[2,1,3,2,4]~~	L=[2,1,3,2,4] L=[1,2,3,2,4] L=[1,3,2,2,4]
9	X	[1,2,3,2,4]	LS	X=1,LS=[2,3,2,4] X=2,LS=[1,3,2,4] X=3,LS=[1,2,2,4] X=4, LS=[1,2,3,2]	X=1,LS=[2,3,2,4] X=2,LS=[1,3,2,4] X=3,LS=[1,2,2,4] X=4,LS=[1,2,3,2] ~~X=2, LS=[1,2,3,4]~~	~~X=1, LS=[2,3,2,4]~~	~~X=1,~~ ~~LS=[2,3,2,4]~~
10	2	[2,1|L]	L	no	~~yes~~	~~yes~~	~~yes~~
11	X	L	[1,3]	L=[X,1,3] L=[1,X,3],X≠1 L=[1,3,X], X≠1,3	~~L=[X,1,3]~~ ~~L=[1,X,3],~~ ~~L=[1,3,X]~~	~~L=[X,1,3]~~	~~L=[X,1,3]~~
12	2	L	LS	L=[2|LS] L=[E,2|T],LS=[E|T], E≠2...	~~L=[2|LS]~~ ~~L=[E|2|T],LS=[E|T]~~ ~~...~~	~~L=[2|LS]~~	~~L=[2|LS]~~
13	X	L	LS	L=[X|LS] L=[E,X|T], LS=[E|T],E≠X, ...	~~L=[X|LS]~~ ~~L=[E,X|T], LS=[E|T]~~ ~~...~~	~~L=[X|LS]~~	~~L=[X|LS]~~

On le voit, malgré le fait que ces trois versions répondent à première vue à la spécification, les implantations proposées répondent très mal à celle-ci. Il est bien sûr possible d'apporter les corrections nécessaires. Par exemple, le test n°3 échoue parce que le second terme de LS n'est pas une liste. On peut solutionner ce problème en exécutant un prédicat de test *list* sur ce second terme. Mais un autre problème se pose : ce test ne réussira que si le terme est clos. Il faudrait alors s'assurer que ce terme est clos au début de l'exécution de la procédure. Il faudrait donc ajouter un nouveau prédicat de

au début de l'exécution de la procédure. Il faudrait donc ajouter un nouveau prédicat de test sur ce terme *et le placer à l'endroit approprié*. En effet, une fois ce test ajouté, la procédure select peut être utilisée avec X et LS variables et L clos *ou bien avec X et LS clos et L variable*. Dans les deux cas, *la relation est vérifiée* mais l'exécution ne réussira pas si L est variable et que le test se fait sur L avant son instanciation à [X|LS]. Et donc *la relation n'est plus vérifiée* ! On le voit, le seul fait de résoudre un problème en introduit beaucoup d'autres, et beaucoup de corrections sont faites pour répondre au modèle d'exécution du langage utilisé[2].

Dans [7], Y. Deville cite plusieurs sortes de problèmes liés à la programmation logique. Ces problèmes sont l'incomplétude, la non équitabilité, l'incohérence, la négation par échec, les informations de contrôle, les opérations extra-logiques et la multidirectionnalité. Ils ont pour origine la distance qui existe entre la sémantique déclarative et le modèle d'exécution du langage Prolog. Ces différents problèmes sont repris ci-après. Nous nous basons sur les exemples donnés dans [7] pour montrer en quoi ils consistent.

3.1.1 Incomplétude de Prolog

Si, par exemple, on définit une procédure p qui calcule la réflexivité et la transitivité de deux termes X et Y, et que l'on considère comme vrai le fait que a et b et que c et d sont réflexifs et transitifs, on a :

```
p(a,b).
p(c,b).
p(X,Z) :-p(X,Y),p(Y,Z).
p(Y,Z) :-p(Z,Y).
```

Si l'on tente de vérifier que :

```
p(a,c)
```

Prolog ne retournera aucune réponse. Prolog utilise une méthode de recherche qui considère les clauses p dans l'ordre dans lequel elles sont rédigées. Or, pour vérifier le but, Prolog doit invoquer la quatrième clause. Mais il ne parvient jamais à cette quatrième clause puisque, après avoir échoué sur les deux premières, il invoque toujours la troisième.

Par rapport à la logique, Prolog est « incomplet » dans le sens où il ne parvient pas à calculer une réponse, alors que cette réponse existe.

3.1.2 Non équitabilité de Prolog

Si on reprend l'exemple de [7] définissant des procédures de concaténation de deux listes et de trois listes, on a les procédures *append* et *append3* suivantes :

---

[2] Alors que la programmation logique veut justement se détacher de ce modèle au moment de la programmation.

```
append([],L,LS).
append([H|T], L, [H|TS]) :- append(T, L, TS).
append3(L1, L2, L3, L) :- append(L1, L2, L12), append(L12, L3, L).
```

Si l'on exécute *append3(X,Y,[2],LS)*, on a la séquence de résultats suivante :

X	*Y*	*LS*
[]	*[]*	*[2]*
[]	*[H1]*	*[H1,2]*
[]	*[H1,H2]*	*[H1,H2,2]*
...		

On voit que X a toujours la même valeur, alors que cette variable peut prendre tout autre forme : le traitement réservé à X n'est pas équitable par rapport au traitement réservé aux autres variables. Le problème se pose quand on exécute *append3(X,Y,[2],LS),X=[1]*, où l'on exécute tout d'abord un appel à append3 afin d'obtenir une valeur pour X et qu'ensuite, on teste si cette valeur est un terme de la forme *[1]*. Cet appel ne réussit jamais, alors que l'appel *append3([1],Y,[2],LS)*, qui est équivalent d'un point de vue logique, va lui réussir.

3.1.3 Incohérence de Prolog

« L'incohérence de Prolog survient quand il calcule une réponse avec succès alors que cette réponse n'est pas une conséquence logique du programme. Les algorithmes d'unification qui ne réalisent pas de tests d'occur-check peuvent donner des solutions incohérentes » [7].

Si l'on reprend l'exemple de la procédure *append* définie ci-dessus, et que l'on exécute *append([],L,[1|L])*, Prolog va invoquer la première clause, calculer une solution et considérer que cette solution est une conséquence logique du programme. Or, la liste [1|L] n'est pas la concaténation de la liste vide et de la liste L.

3.1.4 Négation en Prolog

La négation permet de simplifier fortement l'écriture des programmes Prolog. Mais elle ne correspond pas à une négation logique : son calcul est la négation par échec. La négation par échec essaie dans un premier temps de vérifier la version positive du terme à partir du programme, et si elle n'y parvient pas, elle considère la négation comme vraie. La négation logique, quant à elle, n'a pas besoin de la non présence d'un fait pour affirmer son contraire.

Prolog assure la cohérence de la négation d'un terme en imposant que ce terme soit clos. Mais la complétude de la négation ne peut pas être assurée. Si on a par exemple la procédure :

```
q(a) :-r(a).
q(a) :-not(r(a)).
r(X) :-r(f(X)).
```

où q(a) est manifestement une conséquence logique du programme. Si on exécute le but *q(a)*, Prolog va exécuter infiniment la troisième clause et ne retournera pas de résultat. De nouveau, comme un résultat existant ne peut pas être calculé, le calcul de certaines conséquences logiques (et la négation de celles-ci) peut ne pas se terminer.

3.1.5 L'information de contrôle

Un ensemble de primitives du langage permet de sélectionner les règles à appliquer afin de diriger la recherche lorsque l'on calcule une solution. Cela permet d'appliquer par exemple une stratégie de recherche, ou de choisir l'ordre de sélection d'une clause ou d'un atome. En Prolog, l'opération de base du contrôle est le cut (!).

Ces opérations de contrôle ne sont pas nécessaires pour rédiger des programmes Prolog. Elles ne sont manipulées que dans le contexte d'un modèle d'exécution et ont pour but d'intervenir dans l'algorithme de recherche et de rendre un programme efficace.

Le problème posé par ces opérations est qu'elles ne sont pas gérables uniquement par le compilateur. On doit laisser au programmeur la possibilité de les manipuler. Mais leurs mauvaises utilisations peut « accroître le fossé séparant la sémantique déclarative du modèle d'exécution » [7].

3.1.6 Les opérations extra-logiques

D'autres opérations sont présentes dans le langage, comme les instructions d'entrées-sorties sur divers périphériques. Mais ces opérations trouvent difficilement une définition logique. De plus, elles ont des effets de bord eux aussi difficilement exprimables en logique, ce qui ne fait qu'accroître le fossé entre les deux sémantiques.

3.1.7 Multidirectionnalité

Comme nous l'avons vu dans la partie consacrée à Prolog, les procédures peuvent être invoquées de diverses manières, mais cette caractéristique est rarement utilisée dans la pratique. De plus, il peut être nécessaire de donner un aspect très peu logique à une procédure afin de la rendre réellement multidirectionnelle et d'éviter les problèmes de non terminaison (par exemple, en y introduisant des prédicats de test qui permettent de sélectionner une clause bien précise qui soit capable de s'exécuter sans erreur en fonction des valeurs des paramètres utilisés). Mais les deux restrictions majeures à la multidirectionnalité sont l'efficacité et l'indécidabilité. Une procédure est souvent rédigée en ayant en tête le cas d'utilisation que l'on suppose le plus courant. Mais lorsque cette procédure est utilisée dans « l'autre sens », la recherche d'une variable peut être beaucoup trop longue, voire même incalculable.

Tous ces problèmes font qu'un programme logique correct n'est pas facile à rédiger et que le programmeur ne peut pas se fier à la seule spécification d'une procédure si celle-ci est constamment corrigée au moment de son implantation. Il doit

aussi consulter le code. Il est nécessaire de créer un cadre de travail qui permette non seulement de donner la spécification d'une procédure, mais aussi de spécifier toutes les contraintes qui y sont liées, et cela avant son implantation. De plus, ces spécifications et contraintes doivent être suffisamment sûres pour servir de base (i) à la construction d'une version exécutable qui y réponde de manière satisfaisante, et (ii) à une réutilisation de ces procédures sur seule base de leurs spécifications et contraintes.

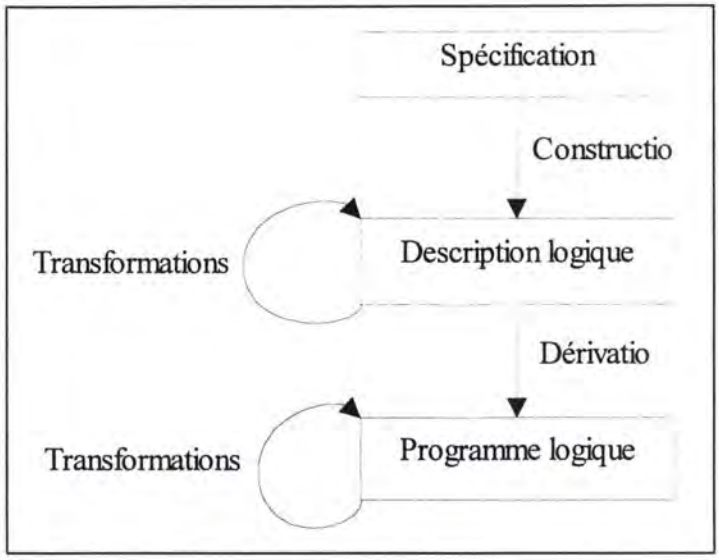
Y. Deville propose de séparer les aspects logiques des aspects non logiques, afin de bénéficier de la puissance de la programmation logique d'une part, et d'un langage logique donné d'autre part. La méthodologie qu'il propose décompose le processus de développement en plusieurs étapes : la spécification de la procédure, la construction d'une description logique de la procédure et la dérivation de cette description logique en une version exécutable dans un langage choisi. La suite de cette partie est consacrée à la présentation de chacune de ces étapes.

3.2 La méthode de Y. Deville

La méthodologie de Y. Deville a pour but la décomposition en plusieurs étapes du processus de développement d'un programme. Cette méthodologie adopte une approche descendante qui permet d'obtenir une version exécutable d'une procédure à partir de sa spécification. Elle opère dans un premier temps à un niveau informel dans lequel on spécifie le problème résolu par une procédure, ainsi que des contraintes qui seront d'application pendant son exécution. A partir de ces spécifications, on construit une première représentation logique de la procédure. Cette représentation est indépendante de tout langage logique[3], et ne contient que des éléments propres à la logique. Enfin, à partir de cette description, on recherche la version exécutable la plus appropriée. C'est à ce niveau que les problèmes liés au langage utilisé sont gérés. Il est également possible de transformer à l'aide d'axiomes une description logique (un programme logique) en d'autres versions équivalentes qui peuvent servir de base à une meilleure construction (dérivation).

[3] Et donc, de tout modèle d'exécution.

La méthode de Y. Deville est schématisée comme suit :



Les trois étapes de cette méthodologie sont la spécification, la description logique et le programme logique. Les opérations de construction et de dérivation permettent de passer respectivement de la spécification à la description logique pour la première, et de la description logique au programme logique pour la seconde. Ces opérations sont obligatoires. Les opérations de transformations sont, quant à elles, optionnelles et permettent de passer d'une représentation à une autre représentation équivalente.

Chacune de ces étapes et de ces transitions sont présentées dans les chapitres suivants.

3.2.1 La spécification d'une procédure

La spécification d'une procédure va définir de manière générale les cas d'utilisation pour lesquels la procédure *doit* se terminer en retournant des résultats corrects, et la signification de ces résultats par rapport aux valeurs en entrée. Les différentes parties d'une spécification jouent un rôle de précondition et / ou un rôle de postcondition. Une précondition est une condition qui doit être vérifiée avant l'exécution de la procédure. Une postcondition est une condition qui doit être vérifiée après l'exécution de cette procédure.

On commence une spécification d'une procédure en identifiant celle-ci et en précisant son arité et ses arguments. On donne ensuite les types de ces arguments, ce qui définit le domaine dans lequel la procédure doit être correcte. La définition de ces types constitue une première précondition sur l'utilisation de la procédure.

Vient ensuite une description, vue par l'utilisateur, du problème résolu par la procédure, ainsi que les directionnalités de celle-ci. Les directionnalités sont les cas d'utilisation que la procédure doit satisfaire. Elles définissent le comportement de la procédure pour chacun des cas d'utilisation, c'est à dire qu'elles précisent la forme que doivent avoir les paramètres avant l'exécution et ce que deviennent ces paramètres une

fois la procédure exécutée. Les types sont de nouveau utilisés dans les directionnalités en tant que préconditions (lorsqu'ils sont utilisés sur les arguments avant l'exécution) et postconditions (lorsqu'ils portent sur les arguments après l'exécution).

Enfin, les éventuels préconditions d'environnement et effets de bord peuvent être définis. Ils ont tous deux trait à la spécification d'éléments extérieurs intervenant dans le comportement de la procédure. Par exemple, une précondition d'environnement est la présence de fichiers ouverts, et les effets de bords sont les écritures effectuées sur ces fichiers.

Les directionnalités sont utilisées parce que peu de procédures sont entièrement multidirectionnelles. Le plus souvent, une procédure est créée dans un cadre d'utilisation donné. Les directionnalités permettent de préciser ce cadre d'utilisation dès la formalisation de la procédure. La description de la procédure et les types des arguments seront utilisés pour construire la description logique. Les types et les directionnalités peuvent, quant à elles, être utilisées pour réaliser des vérifications du programme logique.

Une spécification se fait en langage naturel choisi par le programmeur. La forme générale d'une spécification est la suivante :

L'en-tête de la procédure
Les types des arguments

La description de la procédure vue par l'utilisateur
Les irectionnalités

Les effets de bord de la procédure

Procédure $p(T_1,...T_n)$
Types : T_1 : *type_1*
...
T_n : *type_n*
Relation « ... »

Conditions d'application :
directionnalités
préconditions d'environnement
Forme :
$(C_1 \wedge L_{1,1} \wedge ... \wedge L_{1,m})$
$\vee$...
$\vee (C_k \wedge L_{k,1} \wedge ... \wedge L_{k,m'})$
Effets de bord

D'autres informations pourraient apparaître dans cette spécification. Par exemple, il pourrait y avoir le nombre de solutions retournées par la procédure dans chaque cas d'utilisation prévu. Dans ce cas, on associerait à chaque directionnalité le nombre minimum de solutions retournées par la procédure (nombre entier positif compris entre 0 et ∞[4]) et le nombre maximum de solutions retournées par la procédure (nombre entier positif, ∞ ou *[5]).

[4] ∞ indique un nombre illimité de solutions. On ignore si la procédure cessera de retourner des solutions à un moment donné.
[5] * indique un nombre de solutions qui peut être très grand, mais contrairement à ∞, on sait que la procédure ne retournera plus de solutions à un moment donné.

Par exemple, la spécification de la procédure select/3 serait :

Procédure select(X, L, LS)

Type : X : terme
L : liste
LS : liste

Relation : « X est un élément de L et LS est la liste L dans laquelle on a retiré une occurrence de X ».

Directionnalités :
in(any, ground,any) : :out(ground,ground,ground)
in(ground,any,ground) : :out(ground,ground,ground)

La partie « in » des directionnalités constitue la précondition d'utilisation. La partie « out » constitue la postcondition. Chaque valeur utilisée dans les directionnalités est associée à l'argument correspondant. Dans dans le premier cas d'utilisation, les terme X et LS peuvent correspondre à n'importe quels termes, alors que le terme L doit être clos, c'est à dire que sa composition ne doit contenir aucune variable. Si ces préconditions sont respectées lors d'une exécution de select, les arguments X, L et LS sont tous clos quand l'exécution de select/3 se termine. Dans le deuxième cas, seul le terme L est un terme quelconque, les termes X et LS devant être clos. Au sortir d'une exécution répondant à cette seconde précondition, tous les termes sont de nouveaux clos.

L'information représentée par les directionnalités peut être combinées à celles définies par les types. Ainsi, après toute exécution de select, tous les termes sont clos, mais en plus, L et LS sont des listes. L'utilisation des directionnalités doit respecter les conditions de consistance et de minimalité, de compatibilité et de correction. Une définition précise de ces critères est donnée dans [7].

3.2.2 La construction d'une description logique

Il s'agit ici de *choisir*[6] une représentation logique qui satisfait la spécification d'une procédure donnée. Bien que le choix de cette représentation soit libre, le programmeur peut s'aider de quatre techniques afin de développer une solution. Ces techniques sont le principe d'induction, la généralisation structurelle, la généralisation opérationnelle et la décomposition[7].

Le principe d'induction se base sur le choix d'une relation bien fondée[8] $<$ appliquée sur un ensemble d'objets E et sur une propriété de ces objets. On démontre que la propriété est vraie pour un élément minimal de E[9]. Ensuite on démontre que pour

---

[6] Plusieurs descriptions logiques peuvent être valables pour une même spécification. Il s'agit donc bien d'en choisir une. Comme nous sommes toujours à un niveau logique, aucun critère technique n'intervient dans ce choix.

[7] Une description complète de ces techniques peut être trouvée dans [7].

[8] Une relation binaire $<$ sur un ensemble E est bien fondée si et seulement s'il n'existe pas de suite décroissante infinie d'éléments de E. Une suite décroissante d'éléments de E est une suite telle que $x_1 > x_2 > ... > x_n > ...$

[9] L'élément minimal $e \in E$ est tel qu'il n'existe pas d'autres éléments $e' \in E$ tel que $e' < e$.

un élément e ∈ E non minimal, la propriété est vraie pour un élément f ∈ E < e, et tel que e est construit à partir de f. Dans ce cas, e vérifie aussi la propriété. En généralisant, on a que la propriété est vraie pour tous les éléments de E. On réfléchit par induction sur la structure des termes, et non sur leur type.

Par exemple, < est définie sur les entiers et l'ensemble E est composé d'objets l_i. La relation $l_1 < l_2$ est vérifiée ssi l_1 est un préfixe strict de l_2.

La méthode générale de construction commence par le choix d'un paramètre d'induction et d'une relation bien fondée sur ce paramètre. On détermine ensuite les différents cas qui peuvent se présenter grâce aux différentes formes structurelles du paramètre. Ces cas correspondent aux conditions C_i de la description logique. Enfin, on construit, pour chaque cas, la relation à définir en réduisant le problème à des sous problèmes ou en effectuant des appels récursifs. Ces relations forment les L_i de la description logique. Y. Deville a défini deux méthodes générales pour construire ces relations : la généralisation structurelle et la généralisation opérationnelle. La généralisation structurelle s'effectue sur la structure des données. Elle se base sur le cas minimal, sur la résolution directe d'un cas et sur la décomposition structurelle en problèmes plus simples d'un cas qui ne peut être résolu facilement. La généralisation opérationnelle se base sur l'état général de l'exécution. On connaît la partie déjà examinée (le préfixe de la solution) et celle qu'il reste à exécuter (le suffixe de la solution), et on cherche à réduire ce suffixe en quelque chose « de plus petit » en tirant profit du préfixe. On construit le résultat au fur et à mesure.

3.2.3 La description logique

La description logique est faite dans un langage logique de premier ordre non typé. La forme générale d'une description logique correspond à la définition d'un prédicat « si et seulement si ». Elle est indépendante de tout langage logique et de toute sémantique procédurale. Elle est représentée par :

$$\begin{aligned} \forall X_1,\ldots, \forall X_n : p(X_1,\ldots X_n) \Leftrightarrow \quad & C_1 \wedge (L_{1,1} \wedge \ldots \wedge L_{1,k}) \\ \vee \quad & \ldots \\ \vee \quad & C_m \wedge (L_{m,1} \wedge \ldots \wedge L_{m,p}) \end{aligned}$$

Les conditions C_i définissent la structure des arguments sur laquelle on travaille quand on applique $L_{i,1} \wedge \ldots \wedge L_{i,p}$. Chaque C_i est une conjonction de littéraux, où chaque littéral définit la structure d'un argument. Cette conjonction ne doit pas obligatoirement contenir un littéral pour chaque argument, mais l'ensemble des C_i doit couvrir toutes les utilisations possibles de p. De plus, les C_i doivent être disjoints deux à deux. Les conjonctions de littéraux $L_{i,j}$ sont les relations rendant le prédicat vrai dans le cas de la structure C_j.

Par exemple, la description logique de la procédure select/3 est :

$$\begin{aligned} select(X,L,LS) \Leftrightarrow \quad & L = [] \quad \wedge false \\ \vee \; & L = [H|T] \wedge \; (\quad (H=X \wedge LS=T \wedge list(T)) \\ & \vee \quad (H \neq X \wedge select(X,T,TS) \wedge LS=[H|TS])) \end{aligned}$$

3.2.4 La dérivation d'un programme logique

La dérivation d'un programme logique à partir de sa description logique se fait par transformations syntaxiques successives de formules. Les formules obtenues après transformations doivent être équivalentes aux formules initiales. On commence par créer un ensemble de clauses à partir de chaque description logique. On effectue ensuite un certain nombre d'analyses (sur les types et les directionnalités) afin de déterminer les cas de non terminaison, d'incomplétude, etc. et d'apporter les corrections nécessaires. Par exemple, une analyse détectant une non terminaison de procédure peut rechercher la permutation de clauses ou d'atomes qui entraînera cette terminaison. Suite à ces transformations, il est encore possible de modifier le programme afin d'optimiser l'exécution en ajoutant les cuts appropriés, en tenant compte des effets de bords et du nombre de C.A.S des différents prédicats. Une description complète de toutes ces transformations est donnée dans [7].

3.2.5 Le programme logique

Le programme logique constitue la dernière étape de la méthodologie de Y. Deville. C'est le résultat de la dérivation d'une description logique. Un programme logique est, dans notre cas, rédigé en Prolog mais la dérivation peut viser n'importe quel autre langage logique.

Si on reprend la description logique donnée pour la procédure select/3 et que l'on effectue la dérivation, on obtient la procédure logique :

select(X,L,LS) :- L=[], false.
select(X,L,LS) :- L=[H|T], H=X, LS=T, list(T).
select(X,L,LS) :- L=[H|T], H≠X, select(X,T,TS),LS=[H|TS].

Cependant, l'exécution de Prolog est telle que la première clause peut être supprimée. En effet, si L est une liste vide, les deuxième et troisième clauses ne seront pas vérifiées et le résultat sera « faux ». Or, c'est justement ce que dit la première clause.

Cette procédure peut également être réécrite de façon plus concise en :

select(X,[X|T],T) :- list(T).
select(X,[H|T],[H|TS]) :- H≠X, select(X,T,TS).

3.2.6 Utilisation de la méthode de Y. Deville

La méthode de Y. Deville s'attache principalement à dériver une procédure logique à partir de sa spécification, mais elle n'utilise pas les conditions d'application qui ont été définies pour cette procédure. C'est au programmeur à vérifier si ces conditions sont respectées dans l'implantation obtenue. Cette méthode n'assure qu'une correction partielle des programmes logiques.

L'analyseur exposé dans ce mémoire a pour but de prouver la correction totale des programmes dérivés en utilisant ces conditions de façon automatique. D'autre part, des informations supplémentaires seront données lors de la spécification de la

procédure. L'utilisateur devra définir une ou plusieurs fonctions (ou normes) qui retournent la taille d'un terme sous la forme d'un entier. La comparaison de ces entiers est équivalente à la comparaison de la taille des termes. On pourra ainsi comparer les tailles avant et après l'exécution d'une procédure, et vérifier la terminaison de celle-ci. En effet, la taille d'un terme doit décroître strictement lors d'appels récursifs successifs. Les normes nous permettront aussi d'exprimer le nombre de solutions, non pas sous la forme de bornes inférieure et supérieure, mais à l'aide d'un système d'équations plus complexe dans lequel on pourra mettre en relation la taille des paramètres utilisés et le nombre de solutions. L'analyseur résoudra ce système et vérifiera si les résultats obtenus sont compatibles avec les contraintes sur le nombre de solutions définies dans la spécification. Si ces contraintes sont vérifiées, on saura en plus que l'exécution de la procédure se termine.

On peut par exemple définir la spécification suivante sur la procédure select/3 :

Procédure select(X, L, LS)

Type : X : terme
L : liste
LS : liste

Relation : « X est un élément de L et LS est la liste L dans laquelle on a retiré une occurrence de X ».

Calcul de la taille : $||[X|Y]|| \ ^{10} = 1 + ||Y||$
$||X|| = 0$

Relation « taille » : {L = LS + 1}

Conditions d'application :

in(any, ground, any) : :out(ground, ground, ground)	*{sol = L}*
in(ground, any, ground) : :out(ground, ground, ground)	*{sol = Ls+1}*

Dans cette nouvelle spécification, on voit que le calcul de la taille d'une liste est défini par le programmeur. La norme est récursive et se base sur la structure des termes. Elle sera utilisée pendant la vérification des équations de l'ensemble {L=LS+1}. Cet ensemble définit les relations entre les tailles des termes qui doivent être vérifiées après toute exécution de select/3. L'équation *L=LS+1* impose que, après toute utilisation de la procédure, la taille de L contienne un élément de plus que la taille LS. L'analyseur vérifiera aussi si le nombre de solutions retournées par la procédure[11] et représenté par la variable *sol* respecte bien les conditions qui lui sont imposées.

[10] On n'utilise donc pas la notion de relation bien fondée utilisée par Y. Deville.
[11] Quand elle est exécutée à partir d'une substitution qui satisfait une des classes d'appel définie par les conditions d'application.

4. L'interprétation abstraite

L'analyseur développé dans ce mémoire est basé sur la théorie de l'interprétation abstraite. Cette théorie repose sur quelques principes de base qu'il est nécessaire de définir maintenant, puisque le reste du mémoire se consacre à spécifier un ensemble de notions qui ont toutes un lien direct avec ceux-ci.

Ce chapitre est donc consacré à la présentation de ces principes. Nous verrons ce qu'est un domaine abstrait, ce que sont les opérations abstraites, les fonctions d'abstraction et de concrétisation et en quoi consiste l'algorithme d'interprétation. Nous verrons également qu'il est nécessaire de redéfinir ces notions si l'on veut développer une analyse de programmes rédigés dans un langage particulier.

Nous illustrerons ces notions à l'aide d'exemples réalisant l'interprétation abstraite d'un programme simple. Cette interprétation détermine le signe des données du programme suite à plusieurs additions et soustractions d'entiers. Ces exemples nous montreront que l'interprétation abstraite peut s'appliquer à d'autres types de programmation.

Cette présentation est inspirée sur le cours de Computational Logic de 2eme licence et maîtrise donné par B. Le Charlier.

4.1 INTRODUCTION

L'interprétation abstraite est une méthodologie générale qui nous permet d'obtenir de manière systématique des outils d'analyse statique de programmes.

L'analyse statique cherche à dégager les propriétés sémantiques des programmes. Une propriété sémantique est une caractéristique spécifique qu'un programme possédera lorsqu'il *sera* exécuté. On peut par exemple déterminer si un programme se terminera lorsqu'il *sera* exécuté au départ d'une donnée particulière. Si c'est le cas, nous dirons que « la terminaison du programme » est une caractéristique sémantique.

Pour pouvoir affirmer cela, il ne suffit pas d'exécuter le programme avec la donnée et d'attendre s'il se termine. Il faut savoir s'il se terminera *avant* de l'exécuter. On pourrait aussi vérifier si une procédure calcule un résultat constant, ou bien si elle retourne une infinité de résultats tous différents, ou encore si deux programmes différents réalisent la même chose.

Le théorème d'indécidabilité de Rice nous dit que la recherche de propriétés sémantiques non triviales est un problème indécidable : il ne sera jamais possible de déterminer de façon sûre et automatique une propriété sémantique sur *tous* les programmes pouvant être écrits dans un langage donné, quelle que soit leur complexité.

Cependant, si l'on ne considère qu'un sous-ensemble de ces programmes, il devient possible de rechercher l'*approximation* la plus précise possible des caractéristiques de ces programmes. C'est ce que réalise la méthodologie de l'interprétation abstraite.

L'interprétation abstraite se base sur la définition d'un domaine abstrait et d'opérations abstraites. Elle consiste en l'exécution de chacune des instructions du programme non plus sur le domaine concret du programme[1], mais sur le domaine abstrait que l'on aura défini, et en s'aidant des opérations abstraites que l'on aura également définies. Au fur et à mesure que les instructions sont exécutées, les résultats calculés sur le domaine abstrait approximent de mieux en mieux les propriétés du programme. Comme tout domaine abstrait est un ensemble fini[2], il arrivera un moment où l'approximation des propriétés ne pourra plus être affinée. Donc, une interprétation abstraite se termine (alors qu'une exécution réelle du programme peut ne pas se terminer).

La vérification et l'optimisation de programmes.

Une interprétation abstraite est réalisée avant la phase de compilation d'un programme. Elle peut avoir pour but de capturer les propriétés de ce programme et de *vérifier la correction* de ces propriétés. Si celles-ci sont jugées satisfaisantes, la compilation du programme peut être par exemple refusée.

---

[1] Défini par le langage utilisé.

[2] Une propriété des domaines abstraits est qu'ils sont finis, comme on le verra plus loin.

Par exemple, l'analyse des types en programmation impérative permet de vérifier si les types des paramètres utilisés lors des appels de procédures correspondent bien aux types des arguments définis dans les procédures du programme. Le calcul des invariants en différents points de programme permettrait, entre autres de vérifier si l'utilisation d'un index sur un tableau se fait dans les limites de ce tableau. Dans le cas de la programmation distribuée, il est possible d'analyser les séquences d'instructions entre programmes concurrents et de vérifier l'absence de dead-locks, l'accessibilité aux ressources, etc. D'autres analyses peuvent aussi être faites dans les autres paradigmes de programmation.

Les programmes logiques peuvent eux aussi faire l'objet d'interprétations abstraites, et différents types de vérification peuvent être faites sur base de leurs propriétés. Mais dans le cas de la programmation logique, on peut utiliser d'autres informations. En effet, comme nous l'avons déjà vu dans la partie consacrée à la méthodologie de Y. Deville, les informations fournies par l'utilisateur peuvent elles aussi servir de support à une analyse de validité. Nous n'en dirons pas plus pour le moment, une introduction plus détaillée sur les diverses propriétés intéressantes à calculer pour réaliser ces analyses étant donnée dans le chapitre suivant.

Une interprétation abstraite peut également avoir pour but de capturer les propriétés un programme et d'utiliser celles-ci pendant la phase de compilation afin *d'optimiser ce programme*. Cette solution est très intéressante dans le cas des langages déclaratifs, comme Prolog et ML. Lorsqu'un programmeur utilise de tels langages, il se concentre sur la sémantique des fonctions qu'il veut implanter plutôt que sur la définition d'un fonctionnement bien particulier[3] de ces fonctions. En d'autres mots, ils ignorent les détails d'implantation. C'est au compilateur du langage utilisé de choisir la meilleure représentation exécutable de ces fonctions. Et c'est à ce niveau que l'analyse du programme peut jouer un grand rôle. En effet, la solution la plus simple pour le compilateur d'un langage déclaratif est de générer, pour toutes les procédures d'un programme, un code très général capable de faire face à la majorité des résolutions de fonction rencontrés pendant les différentes exécutions. Mais ce code, parce qu'il est très général, est peu efficace. Si le compilateur dispose d'un ensemble de propriétés calculées avant la phase de compilation, il peut s'aider de ces informations pour détecter quels seront les cas d'utilisation effectifs et générer un code plus spécifique qui réponde à ces seules utilisations.

Le langage Prolog étant un langage déclaratif, ces possibilités d'optimisation peuvent s'y appliquer. Un compilateur Prolog qui ne se base pas sur les propriétés du programme qu'il compile va générer le même code d'unification de termes pour toutes les procédures du programme parce qu'il ignore comment ces procédures seront appelées pendant l'exécution. Ce code va donc répondre à *tous* les cas d'appels possibles de chaque procédure. Ce code a donc toute les chances d'être complexe (et lent). Mais souvent, le programmeur utilise des procédures Prolog en sachant qu'elles ne seront utilisées que d'une certaine manière. Si l'analyseur est capable de repérer ces différents cas d'utilisation, il pourra, pour chacun d'eux, générer le code le plus efficace

[3] La rédaction d'un algorithme qui réalise la sémantique de la fonction.

(rapide) qui y réponde le mieux. Il est ainsi possible d'optimiser des programmes logiques de manière significative, comme nous le verrons dans le chapitre suivant.

Il est évidement possible de combiner la vérification d'un programme avec son optimisation.

La suite de ce chapitre donne une définition des principes de base de l'interprétation abstraite. Ces principes sont le domaine abstrait et les opérations abstraites. On y définit aussi le fonctionnement général de l'interprétation abstraite. Un algorithme simple (et peu efficace) d'interprétation abstraite sera présenté et utilisé dans un exemple réalisant l'analyse des signes de variables entières dans programme particulier.

Les exemples qui suivent supposent que l'ensemble des programmes que l'on veut analyser sont rédigés dans un langage L, et que ce langage définit le domaine des valeurs concrètes C.

4.2 LA DÉFINITION D'UN DOMAINE ABSTRAIT

Un domaine abstrait A est un ensemble fini de valeurs abstraites α qui approximent chacune un ensemble de valeurs concrètes $\{c_1,...,c_n\}$ définies dans un domaine concret C. Chaque valeur abstraite est représentative d'une caractéristique des programmes à analyser. Un domaine abstrait définit donc les propriétés sémantiques que l'on veut calculer. Plus on définit de valeurs abstraites, plus le domaine concret pourra servir de base à une analyse précise.

> *Par exemple, le domaine concret C d'un langage de programmation donné est l'ensemble des entiers* $\aleph$[4]. *L'approximation de ce domaine concret est le domaine abstrait A {+,0 ,-} dont la sémantique abstraite est : « l'élément + représente l'ensemble des entiers strictement positifs, 0 représente l'entier nul, et - représente l'ensemble des entiers strictement négatifs ».*

Remarquons que le choix d'un domaine abstrait est complètement libre et qu'il est souvent réalisé dans un but bien précis. On peut ainsi définir un premier domaine abstrait qui va représenter une caractéristique du programme, et un second domaine qui va capturer une autre caractéristique. Il est même possible de définir un domaine abstrait plus général qui fait interagir les informations fournies par ces deux premiers domaines.

Il existe une correspondance entre les valeurs abstraites et les ensembles de valeurs concrètes qu'elles représentent. Cette correspondance est définie par les fonctions d'abstraction et de concrétisation. La fonction d'abstraction retourne une valeur abstraite représentant les propriétés d'une valeur concrète donnée. La fonction de concrétisation retourne l'ensemble des valeurs concrètes qui satisfont une propriété définie par une valeur abstraite α.

---

[4] Notons que $\aleph$ constitue le « domaine concret » et *ensemble des entiers* donne la « sémantique concrète » de $\aleph$.

Par exemple, les fonctions d'abstraction et de concrétisation pour le domaine abstrait défini ci-dessus sont :

Fonction d'abstraction :

$Abs : \aleph \rightarrow A$

$n \rightarrow +$ si $n > 0$,
$-$ si $n < 0$,
0 si $n = 0$

Fonction de concrétisation :

$Cc : A \rightarrow \aleph$

$- \rightarrow \{n \mid n \in \aleph \wedge n < 0\}$

$0 \rightarrow \{0\}$

$+ \rightarrow \{n \mid n \in \aleph \wedge n > 0\}$

Ces deux fonctions permettront non seulement d'interpréter les résultats de l'analyse dans le domaine concret, mais elles seront aussi utilisées pour vérifier la validation des opérations abstraites définies pour réaliser l'interprétation abstraite.

4.3 LA DÉFINITION DES OPÉRATIONS ABSTRAITES

Les opérations de base du langage L étant définies sur le domaine concret C, il faut définir des opérations correspondantes qui opèrent sur le domaine abstrait. On peut définir ces opérations, dites abstraites, en se basant sur le fonctionnement des opérations concrètes (on simule exactement l'exécution de l'opération concrète mais on redéfinit les résultats sur le domaine abstrait) ou sur leur sémantique (on définit le fonctionnement des opérations abstraites indépendamment du fonctionnement des opérations concrètes, mais les résultats, eux, doivent être consistants). On dira qu'une opération abstraite est consistante si elle approxime de manière sûre l'opération concrète qu'elle redéfinit.

Supposons que les opérations de base de L soient l'addition et la soustraction d'entiers (+ et -). On doit définir, pour chacune de ces opérations, une opération abstraite correspondante. Soient les opérations $+_A$ et $-_A$.

$+_A : A \times A \rightarrow \{A\}$

$\langle -,- \rangle \rightarrow \{-\}$
$\langle -,0 \rangle \rightarrow \{-\}$
$\langle -,+ \rangle \rightarrow \{-,0,+\}$
$\langle 0,- \rangle \rightarrow \{-\}$
$\langle 0,0 \rangle \rightarrow \{0\}$
$\langle 0,+ \rangle \rightarrow \{+\}$
$\langle +,- \rangle \rightarrow \{-,0,+\}$
$\langle +,0 \rangle \rightarrow \{+\}$
$\langle +,+ \rangle \rightarrow \{+\}$

$-_A : A \times A \rightarrow \{A\}$

$\langle -,- \rangle \rightarrow \{-,0,+\}$
$\langle -,0 \rangle \rightarrow \{-\}$
$\langle -,+ \rangle \rightarrow \{-\}$
$\langle 0,- \rangle \rightarrow \{+\}$
$\langle 0,0 \rangle \rightarrow \{0\}$
$\langle 0,+ \rangle \rightarrow \{-\}$
$\langle +,- \rangle \rightarrow \{+\}$
$\langle +,0 \rangle \rightarrow \{+\}$
$\langle +,+ \rangle \rightarrow \{-,0,+\}$

Par exemple, l'addition d'un entier positif et d'un entier nul ne peut donner qu'un entier positif représenté par {+}. Par contre, un entier positif additionné à

un entier négatif peut donner un résultat positif, nul ou négatif ({-,0,+}).
On peut vérifier la consistance de ces deux fonctions à l'aide des fonctions d'abstraction et de concrétisation :

- $\forall c_1, c_2 \in C : Abs(c_1+c_2) \in Abs(c_1)+_A Abs(c_2)$
- $\forall c_1, c_2 \in C : Abs(c_1 - c_2) \in Abs(c_1)-_A Abs(c_2)$
- $\forall a_1,a_2 \in A : \forall c_S \in Cc(a_1+_A a_2) : \exists c_1 \in Cc(a_1), c_2 \in Cc(a_2) : c_S = c_1+c_2$
- $\forall a_1,a_2 \in A : \forall c_S \in Cc(a_1-_A a_2) : \exists c_1 \in Cc(a_1), c_2 \in Cc(a_2) : c_S = c_1-c_2$

4.4 LE FONCTIONNEMENT DE L'INTERPRÉTATION ABSTRAITE

L'algorithme qui va exécuter l'interprétation abstraite d'un programme se base sur le domaine abstrait et les opérations abstraites associés au langage L. Il considère aussi les différents points du programme à analyser et les états associés à ces points.

Un *point de programme* est un endroit dans le programme situé entre deux instructions séquentielles et tel qu'aucune instruction n'est en exécution lorsqu'on se trouve sur ce point. L'exécution[5] d'une instruction fait passer l'état général du programme d'un point à un autre.

L'*état associé à un point de programme* est l'ensemble des valeurs associées aux variables du programme lorsque l'état d'exécution de celui-ci se trouve en ce point. On définit un état pour chaque point du programme à analyser.

L'interprétation abstraite d'un programme consiste en l'exécution des instructions du programmes à l'aide des opérations abstraites, ce qui permet de passer d'un point du programme à un autre, et en la mémorisation, dans l'état associé au point de programme qui suit l'instruction exécutée, des effets de cette exécution sur les valeurs abstraites des variables. Tant qu'un état associé à un point de programme est modifié par l'exécution d'une instruction, il faut réexécuter toutes les instructions précédées de ce point de programme.

Plusieurs algorithmes implantent ce type d'interprétation abstraite : l'algorithme multivariant (le plus simple et le moins efficace), l'algorithme univariant, l'algorithme de O'Keefe, le calcul par point fixe général, etc. Parce qu'il est le plus intuitif pour une introduction aux algorithmes d'interprétation abstraite, on décrira l'algorithme multivariant.

L'interprétation abstraite à l'aide de l'algorithme multivariant.

On suppose que le programme à analyser est constitué d'un ensemble d'instructions, chacune étant précédée d'un point de programme p et suivies d'un ou plusieurs points de programme $p_1,...,p_n$. L'exécution d'une instruction i précédée du point p auquel est associé un état α et suivie des points $p_1,...,p_n$ se note

[5] Ou « l'interprétaion ».

« $(p,\alpha) \rightarrow \{(p',\alpha')\}$ » où $\{(p',\alpha')\}$ est l'ensemble des points de programmes suivant i et pour lesquels les états associés sont modifiés par l'exécution de i.

L'algorithme multivariant se base sur deux ensembles S et R. S est un ensemble de paires (p,α) où p est un point de programme atteint suite à l'exécution d'une instruction i et α est l'état associé à p et modifié par l'exécution de i.. R est l'ensemble des paires (p,α) où p est un point de programme (et α l'état associé à p) atteint suite à l'exécution d'une instruction i et tel que (i) l'instruction suivant p a déjà été réévaluée avec les valeurs de α et (ii) α n'a plus été modifié par l'exécution d'une instruction depuis cette réévaluation.

L'algorithme est :

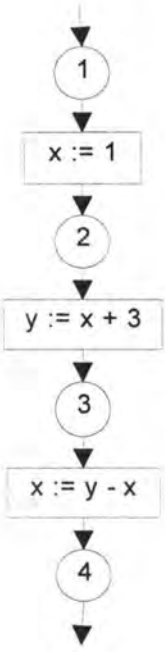
(1) $S := \{p_0,\alpha_0\}$;
(2) $R := \{\}$;
(3) tant que $S \neq \{\}$ faire
(4) choisir $(p,\alpha) \in S$;
(5) $S := S / \{(p,\alpha)\}$;
(6) $R := R \cup \{(p,\alpha)\}$;
(7) $S := S \cup (\{(p',\alpha') : (p,\alpha) \rightarrow \{(p',\alpha')\}\} / R)$

S est construit à partir du point d'entrée p_0 du programme et de l'état α_0 associé à p_0 (1). Tant qu'il y a une paire (p,α) présente dans S (3), il est nécessaire de réeffectuer l'exécution de l'instruction précédée de p à partir des valeurs contenues dans α. (p,α) est alors marqué « en exécution » en étant placé dans R (4),(5) et (6). On exécute ensuite l'instruction suivant p, ce qui donne l'ensemble des points concernés par cette exécution $\{(p',\alpha')\}$. Pour assurer la terminaison de l'algorithme, on ne réinsère pas dans S les paires (p',α') identiques à une paire déjà présente dans R (7).

4.5 EXEMPLE : LA RÈGLE DES SIGNES

A partir des exemples cités auparavant et de l'algorithme multivariant, on va exécuter l'interprétation abstraite du programme ci-dessous afin de connaître le signe des différentes variables en fin de programme. Pour des raisons de simplicité, ce programme est très sommaire.

Le programme à interpréter estle suivant :



Les instructions du langage L sont encadrées et les points de programme sont encerclés.

L'interprétation commence par construire un ensemble $S_1=\{(1,(x/?, y/?))\}$ et $R_1=\{\}$ où « ? » signifie qu'on ignore encore tout du signe des variables x et y. On exécute l'instruction x:=1 en modifiant la valeur abstraite de x et on passe au point de programme 2. On a $S_2 = \{(2,(x/+, y/?))\}$ et $R_2=\{(1,(x/?, y/?))\}$. Pour exécuter l'instruction y := x+3, on calcule $+_A(+,+)$ puisque x est positif dans α associé au point 2 et la valeur entière 3 est positive. Ensuite on assigne à y la valeur abstraite calculée et on passe au point 3. On a $S_3=\{(3,(x/+, y/+))\}$ et $R_3=\{(1,(x/?, y/?)), (2, (x/+, y/?))\}$. On exécute l'instruction x :=y - x en calculant $-_A(+,+)$. La fonction nous renvoie l'ensemble de valeurs abstraites {-,0,+}. On crée l'ensemble S_4 en utilisant chacune de ces valeurs : $S_4=\{(4,(x/-,y/+)), (4,(x/0,y/+)), (4,(x/+,y/+))\}$. $R_4=\{(1,(x/?, y/?)), (2, (x/+, y/?)), (3,(x/+,y/+))\}$. Le point 4 étant le point de sortie du programme, l'algorithme transfert les dernières paires de S_4 dans R_4. Le résultat final est $R=\{(1,(x/?, y/?)), (2, (x/+, y/?)), (3,(x/+,y/+)),(4,(x/-,y/+)),(4,(x/0,y/+)), (4,(x/+,y/+))\}$. En interprétant les résultats, on a que, lorsque le programme se termine, la variable x peut être positive, nulle ou négative tandis que la variable y est toujours positive.

Remarque : l'opération abstraite correspondant à l'assignation d'une valeur à une variable[6] est l'assignation de la valeur abstraite calculée à la variable. Ce qui revient à modifier la valeur abstraite de la variable dans un composant α. Pour cela, nous ne mettons pas de point de programme entre le calcul à droite du symbole d'assignation et l'opération d'assignation en question.

[6] L'assignation d'une valeur entière à une variable constitue une instruction en soi.

5. L'interprétation abstraite de programmes Prolog

Le chapitre précédent présentait de manière générale la méthodologie de l'interprétation abstraite, sans tenir compte d'un paradigme de programmation particulier. Ce chapitre décrit lui aussi une présentation générale de cette méthodologie, mais dans le cas plus particulier de la programmation logique, et encore plus particulièrement dans le cadre de l'utilisation du langage Prolog. On y verra que l'interprétation abstraite permet de réaliser deux grands types d'analyse, le premier visant la correction d'un programme, et le second visant son optimisation. Des exemples illustreront quelques-unes des nombreuses applications que l'on peut construire à partir de ces interprétations. On y définira aussi, de manière très informelle, le type de résultats qu'il est intéressant de calculer pour réaliser l'analyse de programmes logiques. Enfin, l'exemple select/3 sera réutilisé afin d'illustrer les résultats calculables sur un cas concret.

5.1 Introduction

Prolog est le langage logique le plus répandu à l'heure actuelle. Mais pour des raisons d'efficacité, il a été nécessaire d'y introduire des éléments qui le rendent non déclaratif. Par exemple, la recherche d'une solution se base sur un algorithme de

recherche en profondeur d'abord[1]. Ce type de recherche pouvant ne pas se terminer, on a donné la possibilité au programmeur d'intervenir de manière significative dans l'évolution de celle-ci à l'aide de l'instruction « cut ». Cette instruction lui donne la possibilité de stopper la recherche lorsqu'un ensemble de conditions est vérifié. Mais elle n'a rien de logique car la logique ne tient pas compte de la façon dont sera calculé un résultat. Or, « le « cut » agit directement sur *l'algorithme* de recherche du résultat »[7].

D'autres éléments non logiques ont été introduits dans le langage, comme la négation par l'échec[2], les prédicats de test sur les termes (var, list, ground) et la lecture des formules de gauche à droite. L'usage de telles pratiques fait que le programmeur, au lieu de se concentrer sur la sémantique de son programme, se perd dans des contraintes d'implantation.

L'idée est de rendre le compilateur de plus en plus intelligent afin qu'il prenne en charge le maximum de tâches liées à la correction des procédures et au choix des caractéristiques techniques propres à l'implantation de ces procédures. Pour cela, il doit pouvoir capturer suffisamment d'informations sur un programme pour réaliser des analyses « à valeur ajoutée ».

On peut capturer divers types d'information sur un programme. Dans le cadre de la programmation logique, on a entre autres l'analyse des modes, des types et des formes des termes, le partage des variables et de valeurs entre les termes, et l'occur-check[3].

- L'analyse des modes nous permet de savoir si, à un moment donné, un terme peut être unifié à un autre terme. Par exemple, lors d'une unification, une variable peut être unifiée à n'importe quel autre terme, alors qu'un terme clos ne peut être unifié qu'à un terme clos identique. Un terme peut aussi ne plus être une variable, sans pour autant être un terme clos. Dans ce cas, la réussite de l'unification de ce terme dépend du terme avec lequel il sera unifié. Par exemple, le terme f(a,X) ne peut s'unifier qu'à un terme f(a,t) où t est un terme quelconque. L'analyse de son mode peut indiquer dans quels cas cette unification réussit.

- L'analyse des types nous indique si un terme est une liste, ou bien s'il peut être instancié à une liste. Ce type d'information peut, lui aussi, être utilisé pour vérifier le bon fonctionnement d'une unification.

- L'analyse des formes nous donne la structure des termes composés, quel que soit le foncteur. Il permet entre autres de savoir si, avant d'effectuer une unification particulière, un test d'occur-check échouera ou non.

- L'analyse du partage de variables par deux termes indiques si une même variable est présente dans les deux termes à un moment donné. Si c'est le cas et que l'on cherche à unifier ces deux termes, on sait que cette unification ne réussira pas.

---

[1] Une description technique de cet algorithme et des structures sur lesquelles il opère peuvent être trouvées dans [4] et [6].
[2] La négation par échec n'est pas la négation logique « faux ».
[3] Opération qui teste la présence d'une variable dans un terme donné.

- L'analyse du partage des valeurs permet, quant à elle, de dire si deux termes ont déjà été unifiés. Si c'est le cas, le changement de la valeur d'un terme suite à son instanciation vaut aussi pour l'autre terme.

- Enfin, l'analyse de l'occur-check nous montrer si une variable est présente dans un autre terme. Si c'est le cas, l'unification de cette variable et de ce terme ne réussira pas. Cette analyse peut bénéficier des résultats calculés par l'analyse des formes.

D'autres analyses sont réalisables, comme l'analyse de la taille des termes, du déterminisme, du garbage collection, de la détection statique de parallélisme[4], de la spécialisation de programmes, etc. Cependant, elles ne seront pas décrites dans ce mémoire.

Chaque type d'information cité ci-dessus constitue un *domaine abstrait* qui peut être utilisé par un algorithme d'interprétation abstraite. Les informations calculées sur les termes au niveau abstrait sont ensuite interprétées sur des ensembles de termes concrets. Par exemple, si le résultat d'une analyse sur les modes et les types d'un terme t nous dit que t est une liste close, le résultat de toutes les exécutions concrètes du programme retourneront un terme t qui est une liste close. Les *opérations concrètes* et l'*interprétation abstraite* consisteront en une imitation du fonctionnement de Prolog : l'exécution d'une clause procédera de gauche à droite, et l'exécution d'une procédure procédera de la première clause à la dernière.

5.2 OBJECTIFS DE L'ANALYSE DE PROGRAMMES LOGIQUES

L'analyse des programmes logiques peut être réalisée pour satisfaire deux objectifs : la vérification d'un programme, et son optimisation. Elle peut aussi avoir des degrés de précision différents en fonction de la qualité des informations que l'on veut capturer sur un programme. Ce degré de précision dépend de la variété des informations représentables par le domaine abstrait utilisé et de la capacité qu'ont les opérations abstraites à imiter au mieux le calcul exécuté par le programme. Nous présenterons ici une série d'exemples illustrant la vérification et l'optimisation de programmes.

5.2.1 La vérification de programmes logiques

La vérification de programmes peut avoir plusieurs objectifs. Les objectifs qui seront repris ici sont la comparaison d'une procédure par rapport à ses spécifications formelles et à des contraintes extérieures, la vérification de la terminaison du programme, la vérification de la bonne utilisation des opérations non logiques du langage Prolog et la vérification de l'équivalence de deux procédures.

- La première application est la comparaison d'une procédure par rapport à ses spécifications formelles. Nous avons vu dans la partie consacrée à la méthodologie de développement de Y. Deville que l'utilisateur a la possibilité de définir la

[4] Dans le cadre de programmes Prolog concurrents.

sémantique d'une procédure sous une forme purement logique. Il est possible, toujours dans le cadre de cette méthodologie, de dériver de façon automatique une version exécutable de la procédure. Mais le programmeur n'est pas obligé de respecter cette automatisation : il peut lui même écrire une version exécutable. Il est alors intéressant d'exécuter de manière abstraite les deux versions de la procédure et de vérifier si l'ensemble des résultats est identique dans les deux cas.

Toujours dans le cadre de cette méthodologie, le programmeur a la possibilité de définir un ensemble de conditions portant sur chacune des procédures. Il peut par exemple spécifier tous les cas d'utilisation de procédures prévus[5]. La vérification consiste alors à exécuter de manière abstraite chacune des procédures et de vérifier si l'ensemble des résultats concrets répond à ces conditions. Si ce n'est pas le cas, le programmeur peut avoir une description des problèmes rencontrés et ensuite corriger les procédures erronées. Le programmeur a alors une plus grande participation dans le processus de développement des procédures : il peut rédiger des procédures capables de solutionner n'importe quel cas et, dans une application bien spécifique, préciser les seuls cas d'utilisation valides de cette procédure. En combinant cette solution avec l'optimisation de programmes que nous verrons plus loin, il peut constituer des bibliothèques de procédures générales et choisir l'implantation la plus efficace en (re)spécifiant seulement les conditions d'utilisation de chacune de ces procédures dans le cadre d'une application donnée.

- Dans le même cadre, l'utilisateur peut spécifier des contraintes extérieures qui seront d'application lors des exécutions réelles du programme. Ces conditions extérieures peuvent être la complexité du programme la moins tolérable (sous peine de devoir réécrire une version différente de la procédure fautive), ou bien la longueur maximale des chaînes de déférences afin d'optimiser la vitesse d'exécution, ou encore l'espace maximal de travail utilisable pendant l'exécution. L'analyse vérifie si l'exécution du programme respecte ce genre de contraintes.

- L'analyse peut aussi vérifier la terminaison d'une procédure. Si on détecte qu'une procédure ne se termine pas, on peut rechercher de manière automatique et systématique les autres versions de la procédure qui résultent des réarrangements syntaxiques des clauses ou des atomes d'une clause, et de vérifier à nouveau la terminaison suite à chacun de ces réarrangements. On peut ainsi éviter l'utilisation de l'instruction cut.

- On peut aussi simplement vérifier le bon fonctionnement d'une procédure. Par exemple, l'analyse des modes permet de vérifier si la négation par échec est possible sur un terme donné à un moment donné[6]. De même pour l'utilisation des prédicats de test *list* et *ground* : si on détecte que les termes sur lesquels portent ces prédicats ne sont jamais clos au moment du test, on peut de nouveau réarranger de manière syntaxique la clause en déplaçant ces tests à la fin de la clause, moment où ils devront être clos si l'on veut qu'elle réussisse.
  Ce type d'analyse donne la possibilité de rédiger des clauses « réellement » logiques. En effet, une clause qui échoue parce qu'un atome s'exécute sur un terme non clos

[5] En spécifiant des préconditions d'utilisation de la procédure.
[6] La négation par échec ne peut se faire que sur un terme clos.

alors que ce terme sera clos par la suite n'est pas une clause erronée, c'est un problème lié au modèle d'exécution de Prolog (qui exécute les procédures de gauche à droite). Si Prolog considère toutes les inversions d'atomes possibles lorsqu'il exécute une clause, le programmeur n'a plus a se soucier de la manière dont il va considérer la suite des atomes, et il peut rédiger les clauses indépendamment du modèle d'exécution.

- Dans le cas où le programmeur introduit volontairement des éléments non logiques du langage dans le programme qu'il veut vérifier[7], l'analyse va tenir compte du modèle d'exécution de Prolog pour exécuter les opérations non logiques. On peut alors vérifier si le programme se termine, si les cuts se comportent bien de la façon attendue, si les prédicats de test se déroulent correctement, si le nombre de solutions est correct, etc.

- On peut aussi vérifier si une procédure « impure » et une procédure « pure » ont la même sémantique. Elles doivent par exemple se terminer toutes les deux avec un même terme t qui est une liste close, ou bien avec une variable inchangée dans les deux cas. Le programmeur peut ainsi implanter différentes versions de ses procédures et comparer leurs résultats à une version logique de la procédure qu'il sait correcte. Attention que ce type d'analyse ne vérifie pas la sémantique « informelle » de deux procédures mais seulement l'état de leurs termes tout au long de l'exécution.

5.2.2 L'optimisation de programmes

Une interprétation abstraite peut aussi être réalisée afin de capturer suffisamment d'informations sur les exécutions possibles des différentes procédures et de rechercher, sur base de ces informations, la version exécutable la plus efficace du programme.

Il est alors possible de rédiger un programme Prolog « pur », de l'analyser et de dériver automatiquement des versions syntaxiques équivalentes, efficaces, qui intègre des éléments non logiques du langage, et qui peuvent même répondre à un ensemble de paramètres d'exécution comme la vitesse d'exécution, l'occupation de l'espace mémoire, la complexité algorithmique, etc.

Le texte qui suit décrit un ensemble d'optimisations portant sur l'opération d'unification des programmes Prolog.

5.2.2.1 Optimisation de l'algorithme d'unification

Prenons le cas où un programme Prolog cherche à unifier la variable X_1 au terme $f(X_2,...,X_n)$ à partir d'une substitution θ_{in}. La réussite de cette unification dépend des instances de variables présentes dans θ_{in}. Dans les exemples qui suivent, nous préciserons quelles sont ces instances et nous verrons quelles sont quelques-unes des

[7] Les programmes logiques « impurs ».

optimisations que l'on peut apporter à l'algorithme général d'unification mis en oeuvre par Prolog.

L'algorithme général d'unification est la solution standard apportée par Prolog pour pouvoir réaliser une unification en ignorant totalement les instances de variables contenues dans θ_{in} au moment de cette unification. On suppose que les variables $X_1,...,X_n$ présentes dans θ_{in} sont instanciées par n'importe quels termes $t_1,...,t_n$. Avant de résoudre l'unification, Prolog vérifie si les termes sont unifiables (s'ils sont tous les deux clos ou s'ils ont tous deux une forme définie, ils doivent avoir la même forme ; si les deux termes ne sont pas clos, il effectue un test d'occur-check ; etc). Si c'est le cas, il lance l'unification en fonction de la forme des termes unifiés. Dans le cas où il unifie X_1 et $f(X_2,...,X_n)$,il instancie X_1 à un terme $f(Y_2,...,Y_n)$ qu'il crée de toutes pièces. Ensuite, il lance une nouvelle unification pour chacune variables Y_i et des termes t_i.

On le voit, cet algorithme doit très vite réaliser un nombre important d'opérations et est donc coûteux à mettre en oeuvre. Il serait bon de limiter son utilisation aux cas où, vraiment, on ignore tout de la façon dont seront exécutées certaines unifications. Si on parvient à détecter que, lorsqu'on exécute certaines unifications, les termes $t_1,...,t_n$ concernés ne sont pas quelconques, on peut générer un code plus efficace qui remplace l'utilisation de cet algorithme général. C'est ce que l'on va montrer par une série d'exemples où l'on peut obtenir des implantations bien spécifiques qui remplacent l'usage de l'algorithme général tout en assurant le même résultat après exécution. On citera des exemples effectuant une analyse sur les modes, sur les partages de variables et sur la forme. Ces exemples sont extraits de [1].

5.2.2.2 Analyse des modes : exemples d'optimisation

Exemple 5-1
Soient les variables $X_1,...,X_n$ et l'unification $X_1=f(X_2,...,X_n)\theta_{in}$.

- Si on effectue une analyse des modes sur base de l'Exemple 5-1, et que l'on détecte qu'au moment de réaliser l'unification, X_1 est instanciée dans θ_{in} à une variable Y et que les variables $X_2,...,X_n$ sont toutes instanciées dans θ_{in} à des termes clos, cette unification peut être implantée par la suite d'instructions suivante :

  new(X_1,f/n-1) ←*création d'un terme f(...) associé à X_1*
  $X_1[1] := X_2$ ←*on instancie directement le premier argument de f au terme t_2 associé à X_2*
  ...
  $X_1[n-1] := X_n$ ←*on instancie directement le dernier argument de f au terme t_n associé à X_n*

  plutôt que de créer le terme $f(Y_2,...,Y_n)$ associé à X_1 et de lancer (n-1) fois l'algorithme d'unification pour chaque terme Y_i et t_i ($2 \le i \le n$).

- Si l'on détecte par contre qu'au moment de l'unification $X_1=f(X_2,...,X_n)\theta_{in}$, la variable X_1 est instanciée dans θ_{in} à un terme clos et que les variables $X_2,...,X_n$ sont toutes instanciées dans θ_{in} à des termes variables, l'unification revient à tester si X_1 est un terme de la forme $f(t_2,...,t_n)$, et, si c'est le cas, d'instancier les variables

$X_2,...,X_n$ aux termes $t_2,...,t_n$. Si ce n'est pas le cas, l'unification échoue. L'implantation devient :

if struct(X_1,f/n) then
    X_2 := X_1[1]
    ...
    X_n := X_1[n-1]
else
    fail

Le résultat de cette implantation est équivalent à l'exécution de l'unification générale pour chacune des termes $X_2,...,X_n$ et $t_2,...t_n$. Cependant, il ne faut pas que deux variables X_i, X_j ($i \neq j$, $i \geq 2$) aient été unifiées auparavant. Si c'était le cas, on pourrait réaliser l'unification de deux termes alors que ces termes ne sont pas unifiables.

Exemple 5-2
On unifie X_1=X_3, puis on unifie X_3=f(X_1,...,X_n), ce qui revient à unifier X_3=f(X_3,...,X_n). Cette unification n'est pas réalisable à cause de l'occur-check de X_3 dans f.

Cet exemple montre que pour réaliser des optimisations de programme correctes, il faut effectuer une analyse des partages de valeurs en plus de l'analyse des modes, et qu'il faut considérer les résultats des deux analyses pour réaliser ces optimisations.

5.2.2.3 Analyse du partage de variables : exemple d'optimisation

Le partage de variables entre deux termes t_1 et t_2 indique si la variable X_i est présente dans les deux termes t_1 et t_2.

Exemple 5-3
Les termes t_1=[X\|T] et t_2=f(a,X,L) partagent la variable X. Les termes t_1 et t_3=f(H,T,a) ne partagent aucune variable.

L'Exemple 5-2 montre une des utilisations possibles de cette analyse.

Toujours en se basant sur l'Exemple 5-1, si l'analyse des partages de variables détecte que, au moment de réaliser l'unification, le terme $X_1\theta_{in}$ est une variable et que les termes $X_2\theta_{in},...,X_n\theta_{in}$ sont des termes quelconques qui ne contiennent pas X_1, on peut réaliser directement l'unification sans exécuter le test d'occur-check de X_1 dans f($X_2,...,X_n$).

5.2.2.4 Analyse de la forme : exemple d'optimisation

L'analyse de la forme des termes nous renseignent sur le foncteur principal d'un terme t. Toujours dans le cadre de l'Exemple 5-1, si l'analyse détecte que la forme de $X_1\theta_{in}$ est f($u_2,...,u_n$) pour n'importe quelle exécution, que la forme des modes nous dit que X_1 est clos et $X_2,...,X_n$ sont des variables, et que l'analyse des partages de variables

nous dit qu'aucune variable X_i ne partage avec une variable X_j ($2 \leq i,j \leq n$, $i \neq j$), l'unification peut directement être implantée par les instructions suivantes :

$$X_2 := X_1[1], ..., X_n := X_1[n-1]$$

Dans les cas où l'on utilise toujours l'algorithme général d'unification, l'optimisation de programmes est toujours possible. On peut, par exemple, viser la génération automatique d'un nouveau code (source) où l'on introduit des instructions cuts, où l'on remplace des littéraux niés par des cuts, etc. afin d'optimiser la recherche de solutions effectuée par Prolog. On peut aussi programmer de manière statique le choix d'une clause dans un procédure lorsque l'on détecte plusieurs clauses mutuellement exclusives dans la même procédure. Le calcul de la solution évite ainsi l'exécution inutile de clauses. Enfin, il est possible de remplacer toutes les récursivités par des récursivités terminales.

On le voit, le potentiel d'optimisation réalisable par les compilateurs Prolog est énorme à cause de la nature déclarative des langages logiques.

5.3 PRÉSENTATION DE L'INTERPRÉTATION ABSTRAITE DE LA PROCÉDURE SELECT/3

Cet exemple présente les résultats qui peuvent être calculés à l'aide de l'interprétation abstraite sur un programme Prolog, en l'occurrence la procédure select/3[8]. Les informations calculées sur cette procédure sont les modes, les types, les formes, les partages de variables et de valeurs, la taille des termes et le nombre de solutions de la procédure. Si aucun résultat d'un de ces types n'est donné pour un terme t, cela signifie que le résultat de ce type est le plus général possible. Autrement dit, l'interprétation abstraite n'a pas pu retirer une information intéressante pour t.

La procédure select/3 crée une liste LS construite sur base d'une liste L (contenant au moins une occurrence de X) de laquelle on a retiré une occurrence de X. Sa définition est :

select(X,L,LS) :- L = [H|T], H=X, LS=T, list(T).
select(X,L,LS) :- L = [H|T], LS=[H|TS], select(X,T,TS).

Où le prédicat de test *list(t)* teste si le terme t est une liste. Il ne réussit que lorsque t est une liste close, et dans ce cas, il ne réussit qu'une seule fois.

Cette procédure peut être utilisée de diverse manières. Le premier type d'appel que l'on va considérer est celui où X et LS sont des variables distinctes et L est un terme clos quelconque. L'analyse de la procédure est alors le suivant :

---

[8] Seuls les résultats en fin de clause sont présentés ici. Mais il est également possible de fournir les résultats en n'importe quel point de chaque clause. Ce cas est traité dans présentation de la même procédure select/3 dans la partie « Présentation générale de l'analyseur ».

- L'interprétation abstraite de la première clause va nous indiquer que, après n'importe qu'elle exécution réelle de la procédure qui n'échoue pas, L est une liste close non vide, LS est une liste close, X est un terme clos. De plus, L étant un terme clos dès le départ, les unifications L=[H|T] et H=X ne réussissent qu'une seule fois, ainsi que l'exécution du prédicat de test *list*. Donc la première clause ne réussit qu'une seule fois. Enfin, la longueur d'une liste étant calculable, si *sz(L)* est une fonction qui calcule la taille de la liste L, on a que la longueur de LS est égale à sz(L)-1.

- L'interprétation abstraite de la seconde clause commence par exécuter les deux premiers atomes de la clause. Ces atomes ne réussissent qu'une seule fois, et on apprend de leur exécution que L et LS sont tous les deux des termes qui peuvent être instanciés à des listes, et que le premier composant de ces termes est un terme clos. L'interprétation abstraite doit ensuite traiter un appel récursif. Si les conditions d'appel sont respectées (ce qui est le cas dans l'exemple puisque X et LS sont distincts et donc X et TS sont distincts, et que L étant clos, on a T clos), l'exécution de cet appel récursif réussit. On considère dans ce cas que les résultats de cet appel sont les résultats de l'interprétation abstraite de la première clause. On a alors les résultats finaux de l'exécution de cette deuxième clause qui sont que L est une liste close non vide, LS est une liste close dont la longueur est égale à sz(L)-1, X est un terme clos et la clause réussit sz(L)-1 fois (puisqu'il est possible d'effectuer l'appel récursif sur tous les composants de L sauf la tête).

- En regroupant les informations obtenues suite à l'exécution de chaque clause, on obtient les propriétés de la procédure select/3. Ces propriétés sont que, si les conditions d'appels sont respectées, la procédure réussit et on a que L est une liste close non vide, LS est une liste close de longueur égale à sz(L)-1, X est un terme quelconque clos et que la procédure réussit 1 fois lorsque la première clause est utilisée et sz(L)-1 fois quand la deuxième clause est utilisée. Donc la procédure réussit sz(L) fois au total.

Un deuxième cas d'utilisation de la procédure select/3 peut être celui où X et LS constitue une seule et même variable et L est toujours un terme clos quelconque. Dans ce cas :

- L'interprétation de la première clause nous que L est une liste close dont la tête et la queue correspondent à X. Comme L=[X|X], cette clause réussit une seule fois.

- L'interprétation de la deuxième clause nous affirme que celle-ci échoue toujours. Cela est du au fait qu'on appelle select de manière récursive qui rend X et LS différents alors qu'ils doivent être identiques.

- Au total, la procédure sélect réussit une seule fois, et on sait que X est alors clos et L est une liste close de la forme [X|X].

6. Présentation générale de l'analyseur

6.1 Historique de l'analyseur : le système GAIA

6.1.1 Présentation de GAIA

GAIA est un système d'analyse automatique de programmes Prolog. Cette analyse est basée sur l'interprétation abstraite. Comme nous l'avons vu dans le chapitre consacré à ce type d'interprétation, on exécute le programme non pas sur le domaine du langage utilisé, ce qui reviendrait à exécuter le programme pour « voir ce qu'il fait », mais sur un domaine abstrait par lequel il est possible d'exprimer ses propriétés. Dans le cadre de la programmation en Prolog, GAIA calcule l'ensemble des substitutions qui résultent de l'application d'une substitution initiale sur une procédure. Ces substitutions, dites abstraites[1], sont différentes de celles réellement calculées par Prolog. Alors que Prolog réalise une exécution *particulière* du programme et retourne des substitutions dans lesquelles des *termes* sont associés à des variables, GAIA exécute le programme de manière *générale* et manipule des substitutions abstraites dans lesquelles on associe un

[1] Une définition plus complète de ce type de substitution sera donné plus loin.

ensemble de propriétés à chacune de ces variables[2]. Une fois connues, ces substitutions abstraites permettent d'affirmer que lorsque le programme sera exécuté, les termes qui seront calculés à un moment donné satisferont toujours les propriétés calculées par GAIA des mêmes termes, au même point de programme.

Ce système permet donc de capturer la « sémantique abstraite du programme », c'est à dire qu'une fois que l'interprétation est terminée, les propriétés trouvées décrivent l'ensemble des résultats calculables par le programme, c'est à dire, « ce que le programme fait ».

GAIA est un système qui calcule « comme[3] » Prolog mais qui ne calcule pas la même chose. Comme il est basé sur la théorie de l'interprétation abstraite, il a fallu définir son domaine abstrait, c'est à dire l'ensemble des propriétés qui sont capturées. Il a également fallu définir les opérations abstraites qui imitent les opérations de base de Prolog (comme la composition de substitutions, l'unification de deux termes et l'application d'une substitution sur un terme). Enfin, un algorithme abstrait a été créé pour calculer la sémantique abstraite d'un programme qui caractérise la sémantique concrète calculée par Prolog sur ce même programme.

6.1.2 Algorithme générique de GAIA

L'algorithme d'interprétation abstraite de GAIA exécute une procédure (choisie par l'utilisateur) à partir d'une substitution abstraite (donnée par l'utilisateur). Comme on l'a dit ci-dessus, GAIA simule l'exécution de Prolog, mais il ne fonctionne pas comme lui. Pour résoudre un but, Prolog exécute un algorithme de résolution (SLD-resolution) sur un arbre de recherche et utilise des techniques de « marche arrière »[4]. GAIA, quant à lui, résout le but en imitant l'exécution de chaque clause de la procédure correspondante à ce but. Pour cela, il lit les atomes de chaque clause de gauche à droite et utilise la substitution abstraite calculée par l'exécution de chaque atome comme substitution abstraite en entrée pour l'exécution de l'atome suivant. Lorsqu'un atome est un appel de procédure, il lance l'interprétation de celle-ci. Ce n'est qu'une fois cette nouvelle interprétation terminée qu'il passera à l'exécution de l'atome suivant. S'il s'agit d'un appel récursif, il approxime le résultat de celui-ci en appliquant la théorie du point fixe : il exécute la même procédure « jusqu'à ce que plus rien ne bouge ».

Notons que le but de GAIA n'est pas de trouver une substitution qui réponde à un problème. Il ne s'arrête pas quand il a calculé les substitutions réponses. Il exécute *toute* la procédure[5] pour en extraire *toutes* les propriétés.

---

[2] Notons que ces substitutions abstraites sont capables de donner les propriétés des termes associés aux variables présentes dans la substitution, mais aussi les propriétés de chacun des termes qui composent ceux-ci.

[3] Il ne s'agit pas de copier exactement le fonctionnement de Prolog, mais de calculer *le même résultat* transposé sur le domaine abstrait.

[4] Le fonctionnement de Prolog n'étant pas le propos ici, nous renvoyons le lecteur à [4] pour une description plus détaillée.

[5] Et donc *toutes les clauses* de la procédure, ce qui n'est peut être pas le cas lors d'une exécution réelle.

Comme nous l'avons vu dans la partie consacrée à l'interprétation abstraite, il est possible de calculer différentes (combinaisons de) propriétés sur un même programme logique. Il est donc possible de définir différents domaines abstraits. Or, l'utilisation de domaines abstraits différents n'influence pas la manière dont le programme est interprété. Pour cette raison, l'algorithme abstrait de GAIA est un algorithme générique. Il utilise quelques fonctions très générales qu'il est nécessaire de redéfinir quand on veut l'appliquer sur un domaine donné. Il est ainsi possible d'adapter les besoins d'analyse au cas par cas tout en conservant le même schéma d'analyse.

6.1.3 Le domaine abstrait de GAIA

GAIA réalise une analyse du mode, de la forme, du partage de variables et de valeurs pour chacun des termes présents dans une substitution abstraite. Ces différents types d'information constituent le domaine abstrait de GAIA et peuvent être spécifiés comme suit :

- Les modes représentent le niveau d'instanciation des variables du programme à un moment donné. Grâce à eux, il est possible de déterminer si deux termes peuvent être unifiés. Quelques exemples d'utilisation ont été décrits dans la partie « Interprétation abstraite de programmes Prolog ».

- Le partage de variables détecte les interactions existantes entre deux variables, et donc l'absence d'interaction. Il est utile de connaître ces interactions afin de déterminer les effets des unifications et des instanciations.

  Par exemple, la procédure $p(X_1,X_2)$:-$q(X_1),r(X_2)$. Au départ, on a la substitution abstraite $\{X_1$/terme quelconque, X_2/variable$\}$. Supposons que $q(X_1)$ associe un terme clos à X_1. Si on sait[6] que les variables X_1 et X_2 ne partagent pas avant l'exécution de $q(X_1)$, on sait que X_2 reste associé à une variable quand X_1 est devenu clos. Sans cette analyse de partage, on ignore si X_2 est affecté par l'exécution de $q(X_1)$.

- Le partage des valeurs mémorise les unifications réalisées. Si deux termes ont été unifiés, tout changement de valeur d'un terme (suite à son instanciation) vaut aussi pour l'autre terme.

- L'analyse des formes nous donne la structure des termes composés. Elle nous montre l'évolution de la structure de ces termes au fur et à mesure que l'exécution se déroule.

GAIA mémorise ces informations dans des substitutions abstraites. Cette notion est définie (de manière encore très informelle) ci-dessous.

---

[6] Grâce à l'analyse du partage de variables.

6.1.3.1 Substitution abstraite

Une substitution abstraite associe à chaque variable de programme un ensemble de propriétés sur les termes qui pourront être associés à cette variable pendant l'exécution du programme. Si certains de ces termes peuvent être composés, elle associe le même ensemble de propriétés à chacun des sous-termes.

Par exemple, les informations sur les termes contenus dans la substitution concrète {X/a, Y/f(b,[a,b]), Z/f(g(a))} sont représentées dans une substitution abstraite par :

{ X est une variable de programme associée[7] à un terme clos et sa forme est 'a' ;
Y est une variable de programme associée à un terme clos et sa forme est 'f/2'
et les composants de 'f/2' sont 'b' et '[a,b]' ;
Z est une variable de programme associée à un terme clos et sa forme est 'f/1'
et le composant de 'f/1' est 'g(a)' ;
a est un terme clos de la forme 'a' ;
b est un terme clos de la forme 'b' ;
[a,b] est un terme clos de la forme './2' et les composants de './2' sont 'a' et 'b' ;
g(a) est un terme clos de la forme 'g/1' et le composant de 'g/1' est 'a' }

On le voit, une substitution abstraite donne de *l'information* sur tous les termes présents dans la substitution concrète. Elle ne donne pas *les valeurs* de ces termes. Une utilisation plus formelle des substitutions abstraites sera réalisée pendant lors de l'analyse informelle de la procédure select/3 qui se trouve plus loin dans ce chapitre[8].

Une des particularité des substitutions abstraites est qu'on n'utilise par de variables de programme dans les propriétés contenues dans ces substitutions.

Par exemple, la substitution {X_1 a la forme f(X_2), X_2 est une variable nommée Y} n'est pas une substitution abstraite. Mais {X_1 a la forme f(Y), X_2 est une variable nommée Y} est une substitution abstraite.

6.1.4 Les opérations abstraites de GAIA

Une fois le domaine abstrait de GAIA choisi[9], il a fallu redéfinir les opérations génériques de l'algorithme générique pour qu'elles opèrent sur ce domaine particulier. Ainsi, une opération de composition de substitutions abstraites a été définie

---

[7] Nous pourrions aussi dire 'associable' puisque ces informations doivent être calculées avant l'exécution du programme et qu'elles caractérisent les valeurs qui pourront être associées à cette variable.

[8] D'autre part, on ne connaît pas obligatoirement la forme de tous les termes clos. Les informations représentées dans une substitution abstraite peut donc représenter les propriétés de plusieurs substitutions concrètes.

[9] On dit que ce domaine particulier est une instance du domaine générique.

pour calculer les effets qu'ont les compositions de substitutions concrètes sur les informations des termes qui sont composés.

6.1.5 Caractéristiques de GAIA

L'analyse réalisée par GAIA cherche à capturer un nombre suffisant de propriétés sur un programme Prolog pour ensuite l'optimiser. GAIA s'inscrit dans un environnement de programmation tel que les procédures soumises à l'analyse sont *supposées correctes* et pour lesquelles il « reste » à trouver la version exécutable la plus efficace. Le programmeur intervient très peu dans la recherche de la solution : le système lui demande quelles sont les procédures à analyser et quelles sont les propriétés initiales des paramètres de ces procédures. A partir de là, il réalise tous les calculs qui s'avèrent nécessaires pour interpréter complètement les procédures désignées sans faire de nouveau appel à l'utilisateur. GAIA vise l'automatisation des traitements liés à la production de programmes.

D'autre part, pour faciliter la spécification des opérations abstraites et de l'algorithme abstraits, GAIA ne travaille que sur des programmes logiques normalisés[10]. Pour rendre l'utilisation de ces programmes transparente pour l'utilisateur, un module est intégré à GAIA et transforme automatiquement tout programme Prolog en un programme Prolog normalisée équivalent.

6.2 LE NOUVEL ANALYSEUR

L'analyseur développé dans ce mémoire est basé sur GAIA. On réutilise et on étend son domaine abstrait, ainsi que certaines de ses opérations abstraites. Mais surtout, on y représente mieux le déroulement de l'exécution d'une procédure Prolog et on y affine l'information sur les termes ainsi que les relations qui existent entre ces informations.

Le changement le plus important concerne le cadre d'utilisation de l'analyseur. En effet, l'utilisateur va jouer une part plus active dans le déroulement de l'analyse. Il ne donnera plus seulement les données nécessaires au lancement de l'interprétation, mais il donnera aussi toute une série d'informations qui vont valider celle-ci tout au long de son déroulement. Il aura également la possibilité de spécifier les seuls résultats qu'il accepte, tout autre résultat devant entraîner l'arrêt de l'interprétation.

Nous commencerons par voir quelles sont les fonctionnalités ajoutées à GAIA. Nous verrons ensuite que l'utilisation de ces nouvelles fonctionnalités change le principe même de l'analyse. Ce changement va nous pousser à définir un nouvel algorithme générique d'interprétation abstraite qui diffère fondamentalement de celui utilisé dans GAIA. Nous terminerons la présentation par un exemple illustrant le fonctionnement de ce nouvel algorithme.

---

[10] L'annexe « Programmes Prolog normalisés » décrit les caractéristiques de ce type de programme.

6.2.1 Les nouvelles fonctionnalités

L'extension du domaine abstrait de GAIA nous permet de représenter le type des termes et les relations entre les tailles des termes présents dans une même substitution abstraite. L'utilisation de *séquences abstraites* va nous permettre (i) de mémoriser les changements qui surviennent sur les termes au fur et à mesure que l'exécution se déroule et (ii) de représenter *l'ensemble* des substitutions concrètes qui résultent de l'exécution d'une procédure, d'une clause ou d'un atome. Parce qu'il est possible de représenter l'évolution de l'exécution d'une procédure, nous pouvons représenter les relations qui existent entre les tailles des arguments de cette procédure et le nombre de solutions retournées par celle-ci. Nous allons également utiliser une nouvelle substitution abstraite qui nous indiquera, pour une procédure donnée, les cas d'utilisation pour lesquels on est certain d'obtenir au moins une substitution en réponse. Enfin, nous donnons la possibilité au programmeur de spécifier un ensemble de conditions portant sur les procédures du programme, lui permettant ainsi de valider les différentes étapes de l'analyse.

6.2.1.1 Extension du domaine abstrait

Le domaine abstrait de cet analyseur permet de représenter non seulement les informations déjà définies dans GAIA (modes, formes, partages de variables et de valeurs), mais aussi le type des termes et les relations qui existent entre les tailles des termes.

L'analyse du type des termes, couplée à l'analyse des modes, donne une meilleure approximation de l'ensemble des termes qui peuvent être associés à une variable du programme. Par ce que ces deux analyses sont complémentaires, elles peuvent chacune tenir compte des informations capturées par l'autre pour améliorer la précision de leurs calculs. Or, comme nous l'avons vu lors de l'introduction à l'interprétation abstraite de programmes Prolog, plus les propriétés d'un programme sont précises, plus la vérification et l'optimisation de ce programme peuvent être efficaces.

D'autre part, les informations sur les tailles des termes peuvent être utilisées pour des analyses de terminaison et de complexité. Ces tailles seront représentées par des systèmes d'(in)équations linéaires définis sur les variables de programme.

Tous ces types d'information peuvent être représentés à l'aide d'une substitution abstraite. On va donc réutiliser et compléter cette notion déjà présente dans GAIA.

Encore une fois, parce que le domaine abstrait est redéfini, il est également nécessaire de revoir les opérations abstraites grâce auxquelles on va pouvoir approximer au mieux un programme Prolog. Nous verrons que ces opérations peuvent combiner les différents types d'information représentés par le domaine pour réaliser les calculs imitant au mieux le comportement réel du programme.

6.2.1.2 Les séquences abstraites

Lorsque Prolog exécute une procédure et que cette exécution se termine, il retourne une substitution concrète en réponse. Mais il peut aussi retourner une séquence de substitutions (lorsqu'il y a plusieurs résultats possibles pour une même procédure). Et inversement, quand l'exécution de la procédure ne se termine pas, il ne retourne aucune substitution.

On notera $\langle\theta_{in},S\rangle$ la séquence de substitutions retournée par l'exécution d'une procédure à partir de la substitution en entrée θ_{in}. Nous noterons par θ_i une substitution présente dans S.

Une présentation des différentes séquences de substitutions qui peuvent être retournées par Prolog est donnée ci-dessous. Une présentation intuitive du calcul de ces séquences est donnée un peu plus loin. La partie suivante décrit une nouvelle structure qui nous permet de représenter des séquences de substitutions au niveau abstrait. Enfin, un exemple concret illustre une de ces séquences au niveau concret et au niveau abstrait.

6.2.1.2.1 Les différentes séquences de substitutions

Quand une procédure est exécutée à partir de θ_{in}, Prolog retourne des séquences de substitutions. Ces séquences peuvent être caractérisées comme suit :

1) si Prolog retourne une seule substitution de réponse θ_1, on note la séquence $\langle\theta_{in},\langle\theta_1\rangle\rangle$[11] ;

2) si Prolog retourne une succession de substitutions réponses $\theta_1,...,\theta_n$ puis, ayant calculé toutes les réponses possibles, il termine l'exécution de la procédure, on note la séquence $\langle\theta_{in},\langle\theta_1,...,\theta_n\rangle\rangle$. On dira que cette séquence est finie ;

3) si Prolog cherche « infiniment » une première substitution réponse, on note la séquence $\langle\theta_{in},\langle\perp\rangle\rangle$[12]. On dira que cette séquence est incomplète ;

4) si Prolog retourne une succession de substitutions réponses $\theta_1,...,\theta_n$ puis *chercher infiniment la réponse suivante*, on note la séquence $\langle\theta_{in},\langle\theta_1,...,\theta_n,\perp\rangle\rangle$. On dira que cette séquence est infinie ;

5) si Prolog retourne une succession infinie de substitutions réponses $\theta_1,...,\theta_n,...$, on note la séquence $\langle\theta_{in},\langle\theta_1,...,\theta_n,...\rangle\rangle$.

6.2.1.2.2 Méthode de calcul de séquences de substitutions

Il s'agit ici de présenter de manière intuitive une méthode de calcul des séquences de substitutions différente de celle utilisée par Prolog. En effet, cette méthode n'utilise ni l'arbre de recherche ni les techniques de « marche arrière » propres à Prolog.

---

[11] Cette situation constitue en fait un cas particulier de la situation n°2.
[12] Cette situation constitue en fait un cas particulier de la situation n°4.

Cependant, nous verrons que les résultats calculés ici sont équivalents à ceux de Prolog. Cette méthode est importante car elle est la base de l'interprétation abstraite qui va être utilisée dans l'analyseur.

Pour commencer cette présentation, on va utiliser une clause cl de la forme $H :-L_1,...,L_n$. On veut approximer le calcul concret suivant : la tête de H de la clause est appelée avec une substitution d'appel θ_{in}. Cette substitution est définie sur les variables de tête de la clause cl. Elle est tout d'abord étendue à toutes les variables de cl, ce qui donne la substitution θ_{in}'. On crée alors une première séquence de substitutions $S_0 = \langle\theta_{in},\langle\theta_{in}'\rangle\rangle$. θ_{in}' est donc la première substitution créée depuis le début de l'exécution de cl.

Supposons que $S_{i-1} = \langle\theta_{in}',\langle\theta_1,..., \theta_n\rangle\rangle$ soit la séquence de substitutions calculée avant l'exécution du littéral L_i. L'exécution de ce littéral se déroule en trois étapes.

La première étape consiste en la projection de chaque substitution θ_j de S_{i-1} sur les variables utilisées dans L_i. On obtient la séquence de substitutions *restreintes* $\langle\theta'_1,..., \theta'_n\rangle$. La seconde étape consiste en l'exécution de L_i à partir de chacune des substitutions θ'_k ($1\leq k\leq n$). Chacune de ces exécutions retourne la séquence de substitutions $\langle\theta'_k,\langle\theta'_{k_{1_k}},..., \theta'_{k_{m_k}}\rangle\rangle$. La troisième étape consiste en la réinsertion des termes associés aux variables non utilisées dans L_i dans chacune des substitutions $\theta'_{k_{l_k}}$ ($1\leq l\leq m$), *et à propager les effets de l'exécution de L_i sur les termes qui étaient liés aux variables utilisées dans L_i*[13]. Cette troisième étape retourne, pour chacune des exécutions de L_i, une séquence résultat $\langle\theta_k,\langle\theta_{k_{1_k}},...,\theta_{k_{m_k}}\rangle\rangle$ ayant toutes même domaine que θ_{in}'.

Toutes ces séquences sont alors concaténées pour donner la séquence $S_i=\langle\theta_{in}',\langle\theta_{1_{1_1}},...,\theta_{1_{m_1}},...,\theta_{n_{1_n}},...,\theta_{n_{m_n}}\rangle\rangle$. Cette séquence représente tous les résultats calculés par la procédure depuis l'exécution du premier littéral L_1 à partir de θ_{in}' jusqu'à l'exécution du littéral L_i.

L'exécution du littéral L_n renvoie la séquence résultat S_n. Chaque substitution présente dans S_n est alors restreinte aux variables de la tête de la clause, ce qui donne la séquence S_{out}. Cette dernière séquence représente les substitutions réponses retournées par l'exécution de la clause cl à partir de θ_{in}.

On peut schématiser ce calcul par :

$$(\theta_{in})\ H :- (S_0)\ L_1, (S_1)\ L_2,..., (S_{i-1})\ L_i, (S_i)\ L_{i+1},..., (S_{n-1})\ L_n. (S_n)\ (S_{out})$$

où l'exécution de chaque littéral L_i est décomposée en trois étapes (restriction aux variables utilisées dans un littéral, exécution du littéral, réintégration des termes non utilisés dans le littéral) et où θ_{in} et S_j ($0\leq j\leq n$) sont les résultats calculés par la méthode décrite ci-dessus.

---

[13] Nous verrons plus loin ce que signifie cette remarque.

termes non utilisés dans le littéral) et où θ_{in} et S_j ($0 \leq j \leq n$) sont les résultats calculés par la méthode décrite ci-dessus.

Cette présentation est importante car elle donne un premier aperçu du fonctionnement de l'analyseur, c'est à dire de la manière dont on va simuler l'exécution d'un programme Prolog. On le voit, cette méthode est très différente du calcul réellement effectué par Prolog, mais si on compare ses résultats avec ceux retournés par Prolog, ils sont équivalents[14].

6.2.1.2.3 Les séquences abstraites

Les substitutions abstraites définies dans GAIA sont insuffisantes pour représenter l'information sur une *séquence* de substitutions : elles ne peuvent donner de l'information que sur chaque substitution θ indépendamment des autres.

Pour représenter ces différentes séquences concrètes, on va utiliser la notion de *« séquence abstraite »* proposée par B. Le Charlier et al. Une séquence abstraite donne *l'information* sur la substitution utilisée lors de l'appel d'une procédure et sur la séquence de réponses retournées par la résolution de cet appel. L'information donnée par une séquence abstraite définit les propriétés des séquences S_i décrites ci-dessus. On note une séquence abstraite $\langle \beta_{in}, \langle B \rangle \rangle$ où β_{in} est la substitution abstraite en entrée utilisée pour lancer l'exécution d'une procédure, et où B définit les propriétés qui résultent de l'exécution de la procédure à partir de β_{in}[15].

Le calcul des séquences abstraites est similaire au calcul des séquences décrit ci-dessus : la substitution θ_{in} est approximée par une substitution abstraite β_{in} et chaque séquence S_i est approximée par une séquence abstraite B_i. On a :

$$(\beta_{in})\ H :- (\mathbf{B_0})\ L_1, (\mathbf{B_1})\ L_2, \ldots, (\mathbf{B_{i-1}})\ L_i, (\mathbf{B_i})\ L_{i+1}, \ldots, (\mathbf{B_{n-1}})\ L_n. (\mathbf{B_n})\ (\mathbf{B_{out}})$$

Encore une fois, ce schéma nous donne un aperçu du fonctionnement de l'analyseur au niveau abstrait. Et encore une fois, l'exécution de chaque littéral se déroule en trois étapes (on projette les propriétés définies par B sur les variables utilisées dans L_i, on exécute L_i sur la séquence restreinte, puis on réintègre les propriétés sur les termes non utilisées dans L_i et on propage les effets de l'unification sur les propriétés de ces termes).

Les avantages que l'on trouvera dans l'utilisation des séquences abstraites sont :

---

[14] Toutes les solutions retournées par cette méthode d'exécution seront aussi retournées par le calcul de Prolog, et vice versa.

[15] La composition de B est libre. Elle ne représente pas nécessairement une *suite* de substitutions abstraites. Il est en effet possible de représenter *les propriétés* de toutes les substitutions concrètes (qui résultent de l'exécution de la procédure à partir de toutes les substitutions concrètes représentées par β_{in}) par une structure à taille fixe (une suite de substitutions abstraites étant par définition de taille variable). Comme nous le verrons, l'analyseur ne représente pas une séquence de substitutions par une séquence de substitutions abstraites, mais bien à l'aide de deux substitutions seulement. Ceci assure entre autres sa terminaison. **Le terme *séquence* utilisé dans « séquence abstraite » est donc purement conventionnel.**

- Comme il est possible d'utiliser des substitutions abstraites dans une séquence abstraite, on peut y représenter les modes, types, partages et relations entre les tailles aux différentes étapes de l'exécution ;

- Comme chaque séquence abstraite contient la substitution abstraite en entrée β_{in} et les propriétés des termes pendant et en fin d'exécution, on pourra comparer l'information sur les variables à des moments différents[16] ;

- Il sera possible d'exprimer le nombre de solutions de l'exécution de L_i par rapport à la taille des paramètres en entrée de l'exécution de ce littéral. Cette information sera représentée par un système d'(in)équations définies sur les variables de la séquence abstraite et sur une variable supplémentaire, *sol*, qui représentera le nombre de solutions de la séquence abstraite. Un tel système permet de définir des relations complexes entre les termes entre eux, et entre les termes et le nombre de solutions.

- Il sera possible de réaliser une analyse de déterminisme sur base du nombre de solutions retournées par la procédure. Cette analyse consiste à vérifier si une procédure retourne 0 ou 1 solution. Si un programme ne possède que des procédures déterministes, il est possible de l'optimiser de manière significative.

6.2.1.2.4 Exemple de calcul d'une séquence de substitutions concrètes

Cet exemple a pour but d'illustrer le calcul des séquences de substitutions à l'aide de l'algorithme décrit ci-dessus. Pour cela, on suppose le code Prolog suivant :

```
p(X,Y,Z) :- X=a, Y=f(b), q(X,Y,Z), W=Z.
q(X,Y,Z) :- Z=[X|Y].
q(X,Y,Z) :- Z=[Y|X].
```

La procédure *p* va être exécutée à partir d'une substitution θ_{in}={H/T, Y/T} où H et Y partagent donc la variable T. L'exécution va calculer successivement S_0, S_1, S_2, S_3, S_4 et θ_{out} et peut être schématisée par :

(θ_{in}) *p(X,Y,Z) :-* (S_0) *X=a,* (S_1) *Y=f(b),* (S_2) *q(X,Y,Z),* (S_3) *W=Z .* (S_4) (S_{out})

La première opération consiste à étendre le domaine de θ_{in} à toutes les variables de p. On obtient la première séquence S0.

$S_0=\langle\theta_{in}, \langle\theta_{in}'\rangle\rangle$ *où* $\theta_{in}' = \{H/T, Y/T\}$

L'exécution du littéral *X=a* commence par la restriction de θ_{in}' à la variable X. On obtient $\theta_{in,1}'=\{\}$. On exécute le littéral à partir de $\theta_{in,1}'$ et on obtient la séquence $S_1'=\langle\theta_{in,1}',\langle\{X/a\}\rangle\rangle$. On y réinsère les associations définies dans θ_{in}' des variables non utilisés dans le littéral, et on obtient S_1.

---

[16] Ce qui permet de prouver la (non) terminaison d'une procédure. Par exemple, la seule connaissance de B_{i-1} permet de prouver la terminaison d'un appel récursif L_i puisqu'on pourra comparer la norme des arguments utilisés comme paramètres à leur norme au début de l'exécution de la clause (un appel récursif se terminant si l'on atteint un cas terminal résolvable).

S_1=⟨θ_{in}', ⟨θ_1⟩⟩ où θ_1 = {X/a, H/T, Y/T} est obtenue par unification de X et a dans θ_{in}'.

L'exécution du littéral *Y=f(b)* commence par la restriction de θ_1 à la variable Y. On obtient $\theta_{in,2}$'={Y/T}. On exécute le littéral à partir de $\theta_{in,2}$' et on obtient la séquence S_2'=⟨$\theta_{in,2}$',⟨{Y/f(b)}⟩⟩. On y réinsère les associations définies dans θ_1 des variables non utilisés dans le littéral, on propage les effets de l'unification (ici, H doit lui aussi être associé à f(b)) et on obtient S_2.

S_2=⟨ θ_{in}', ⟨θ_2 ⟩⟩ où θ_2 = {X/a, Y/f(b), H/f(b)} est obtenue par unification de Y et f(b) dans θ_1.

L'exécution du littéral *q(X,Y,Z)* commence par la restriction de θ_2 aux variables X,Y,Z. On obtient $\theta_{in,3}$'={X/a,Y/f(b)}. On exécute le littéral à partir de $\theta_{in,3}$' et on obtient la séquence S_3'=⟨$\theta_{in,3}$',⟨θ_3'={X/a,Y/f(b),Z/[a|f(b)]}, θ_4'={X/a,Y/f(b),Z/[f(b)|a]}⟩⟩. On réinsère dans θ_3' et θ_4' les associations définies dans θ_2 des variables non utilisées dans le littéral, on propage les effets de l'unification (ici, aucun) et on obtient S_3.

S_3=⟨ θ_{in}', ⟨θ_3, θ_4⟩⟩ où θ_3={X/a, Y/f(b), Z/[a|f(b)], H/f(b)} est obtenue par l'exécution de l'appel q(X,Y,Z) *dans θ_2, ainsi que θ_4={X/a, Y/f(b), Z/[f(b)|a], H/f(b)}.*

On voit qu'à partir de la substitution θ_2, l'exécution de l'appel de procédure retourne deux substitutions réponses différentes.

S_4 est obtenu par l'exécution du littéral *W=Z*, une permière fois avec θ_3 et une seconde fois avec θ_4. Le raisonnement est identique à celui définit ci-dessus. On obtient la séquence S_4.

S_4=⟨θ_{in}', ⟨θ_6, θ_7⟩⟩
où θ_6={X/a, Y/f(b), Z/[a|f(b)], W/[a|f(b)], H/f(b)} est obtenue par l'unification de W et Z dans θ_3 et θ_7={X/a, Y/f(b), Z/[a|f(b)], W/[f(b)|a], H/f(b)} est obtenue par l'unification de W et Z dans θ_4.

S_{out} doit être la représentation des substitutions contenues dans S_4 mais restreinte aux variables de la tête de la clause.

S_{out}=⟨θ_{in}, ⟨{X/a, Y/f(b), Z/[a|f(b)]}, {X/a, Y/f(b), Z/[a|f(b)], H/f(b)}⟩⟩.

On voit que l'ensemble des résultats calculés par notre algorithme calcule le même ensemble de résultats que le ferait Prolog. On voit également que le nombre de solutions retournées par p est égal au nombre de solutions retournés par la procédure q. De plus, dans chacune des substitutions contenant les variables de programme Z et W, on pourrait préciser les relations entre les tailles des termes associés aux variables de programme. On pourrait avoir, par exemple :

sz(Z)=sz(W)=sz(Y)+1

où sz est la fonction qui calcule la taille d'un terme. On voit que si on construit un terme à partir de Y, alors la taille de ce terme construit est supérieure de 1.

6.2.1.2.5 Exemple de calcul de séquences abstraites

Nous allons ici reprendre l'exemple précédent, mais en réalisant les calculs sur un domaine abstrait calculant les modes, les formes, les partages de valeurs pour chacun des termes associés aux variables de programme, ainsi que le nombre de solutions des séquences abstraites. Deux variables partagent une valeur dès qu'elles sont unifiées. Elles partagent une variable si les informations qui leurs sont associées contiennent une variable commune.

On choisit de représenter le composant B d'une séquence abstraite par une séquence de substitutions abstraites[17]. La procédure *p* est exécutée à partir d'une substitution abstraite β_{in} représentant l'information pour chacune des variables de programme de p.

β_{in}={X/variable, Y/variable T, Z/variable, H/variable T} où Y et H partagent la même variable T.

L'exécution va calculer successivement B_0, B_1, B_2, B_3, B_4 et B_{out} que l'on schématise par :

*(β_{in}) p(X,Y,Z) :- **(B_0)** X=a, **(B_1)** Y=f(b), **(B_2)** q(X,Y,Z), **(B_3)** W=Z . **(B_4)** **(B_{out})***

La première opération réalisée est l'extension de β_{in} à toutes les variables de la clause et à partir de laquelle on construit B_O.

B_0=⟨β_{in}, ⟨β_{in}'⟩⟩ où β_{in}'={X/variable, Y/variable T, Z/variable, H/varible T, W/variable}.

On suppose que la taille des termes peut être calculée par la norme ||.|| telle que ||t||=0 si t n'est pas un terme de la forme $[t_1,t_2]$, et $||[t_1|t_2]||=1+||t_2||$.

L'exécution des deux premiers littéraux commence par la restriction des composants des séquences abstraites qui précédent ces littéraux aux variables utilisées dans ces littéraux (noté *restr*), puis on exécute le littéral sur la séquence abstraite restreinte (noté *exe*), et ensuite on réintègre[18] les informations sur les variables non utilisées dans ces littéraux et on propage les effets des unifications sur les propriétés de ces variables (noté *ext*).Nous donnons directement les séquences abstraites calculées.

<u>Exécution du premier littéral.</u>

Restr : $\beta_{1,restr}$ = {X/var}
Exe : B_1'=⟨$\beta_{1,restr}$, ⟨{X/clos de la forme a}⟩⟩
Ext : B_1=⟨β_{in}', ⟨β_1⟩⟩ où β_1 est obtenue par unification de X et 'a' dans β_{in}' :
β_1 = {X/clos de forme 'a', Y/variable T, Z/variable, W/variable, H/variable T} et où les tailles de tous les termes sont toujours égales à 0 et le nombre de solutions est {sol=1}.

---

[17] Ce n'est pas la manière de faire retenue pour l'analyseur, mais elle nous évite de déjà introduire des notions trop complexes.
[18] On dira aussi qu'on étend la séquence abstraite aux variables non présentes dans le littéral.

Exécution du second littéral.

Restr à partir de β_1 : $\beta_{2,restr}$ = {Y/variable T}
Exe : B_2'=⟨$\beta_{2,restr}$,⟨{Y/clos de forme f(b)}⟩⟩.
Ext : B_2=⟨ β_{in}', ⟨β_2 ⟩⟩ où β_2 est obtenue par unification de Y et f(b) dans β_1 :
*β_2 = { X/clos de forme 'a', Y/clos de forme 'f(b)', Z/variable, W/variable, **H/clos de forme f(b)**}*[19]
et où les tailles de tous les termes sont toujours égales à 0 et le nombre de solutions est {sol=1}.

Le calcul du résultat de l'appel à la procédure q se fait comme suit. On exécute chaque clause de q avec chaque substitution abstraite contenue dans B_2 restreinte aux variables X, Y, Z utilisées pour l'appel (la substitution abstraite utilisée en entrée pour l'exécution de ces deux clauses est $\beta_{in,q}$={X/clos de forme 'a', Y/clos de forme 'f(b)', Z/variable}).

La première clause va retourner une séquence ⟨$\beta_{in,q}$, ⟨$\beta_{out,q,1}$ = {X/clos de forme 'a', Y/clos de forme 'f(b)', Z/clos de la forme [a|f(b)]}⟩⟩. La seconde clause va retourner la séquence abstraite ⟨$\beta_{in,q}$, ⟨$\beta_{out,q,2}$ = {X/clos de forme 'a', Y/clos de forme 'f(b)', Z/clos de la forme [f(b)|a]}⟩⟩. Le résultat de la procédure q consiste alors en la concaténation de ces deux séquences abstraites, c'est à dire ⟨$\beta_{in,q}$, ⟨{X/clos de forme 'a', Y/clos de forme 'f(b)', Z/clos de la forme [a|f(b)]}, {X/clos de forme 'a', Y/clos de forme 'f(b)', Z/clos de la forme [f(b)|a]}⟩⟩. Chacune de ces substitutions abstraites est de nouveau étendue à toutes les variables de la clause p. Le résultat est B_3.

B_3=⟨ β_{in}', ⟨β_3, β_4⟩⟩ où β_3 et β_4 sont obtenues par l'exécution de l'appel q(X,Y,Z) *dans β_2 :*
β_3= {X/clos de forme 'a', Y/ clos de forme 'f(b)', Z/clos de forme [a/f(b)], W/variable, H/clos de forme f(b)} et ||Z||=1 ;
β_4= {X/clos de forme 'a', Y/ clos de forme 'f(b)', Z/clos de forme [f(b)|a], W/variable, H/clos de forme f(b)} et ||Z||=1 ;

Le nombre de solutions dans B_3 est exprimé par le système {(||Y||$_{\beta_3}$+1)+(||Y||$_{\beta_4}$+1)=2}

L'exécution de W=Z se fait à partir de β_3 et β_4, ce qui donne β_6 et β_7.

B_4=⟨β_{in}', ⟨β_6,β_7⟩⟩ où β_6 est obtenue par l'unification de W et Z dans β_3 et β_7 est obtenue par l'unification de W et Z dans β_4 :
β_6={X/clos de forme 'a', Y/ clos de forme 'f(b)', Z/ clos de forme [a/f(b)], W/clos de forme [a\|f(b)], H/clos de forme f(b)} et ||Z||=||W||=1 ;

[19] Cette substitution abstraite doit aussi être capable de renseigner sur le terme b, qui compose le terme f(b) associé à Y. Comme il s'agit ici d'une présentation intuitive, on n'encombre pas la substitution abstraite avec ces informations mais on doit garder cette idée à l'esprit.

β_7={X/clos de forme 'a', Y/clos de forme 'f(b)',
Z/ clos de forme [f(b)|a], W/clos de forme [f(b)|a] , H/clos de forme f(b)}
et ||Z||=||W||=1.

Le nombre de solutions dans B_4 est exprimé par le système {(||Y||$_{\beta 6}$+1)+(||Y||$_{\beta 7}$+1)=2}. Les variables Z et W dans ces deux substitutions abstraites partagent la même valeur.

B_{out} est la représentation abstraite des substitutions abstraites contenues dans B_4 mais restreinte aux variables de la tête de la clause.

B_{out}=(β_{in},{ {X/clos de forme 'a', Y/ clos de forme 'f(b)', Z/ clos de forme [a/f(b)], H/f(b) }, {X/clos de forme 'a', Y/clos de forme 'f(b)', Z/ clos de forme [f(b)|a], H/f(b)}})

Ces informations nous affirment que, après toute exécution de la procédure p, X sera toujours un terme clos de la forme 'a', c'est à dire une constante, alors que Y sera un terme composé de la forme 'f(b)'. Z, quant à elle, sera toujours associée à un terme clos, mais la forme de ce terme peut être '[a|f(b)]' ou '[f(b)|a]. Ces résultats sont confirmés par l'algorithme appliqué au cas concret. De plus, les informations associées à H ont été modifiées par cette exécution.

Quant au nombre de solutions (noté *sol*) de la procédure p, on a {sol=1} pour les séquences B_0, B_1 et B_2. La procédure q ayant quant à elle un nombre de solutions égal à 2, les séquences abstraites B_3 et B_4 ont elles aussi {sol=2}. Le nombre de solutions de la procédure est lui aussi égal à 2.

Ceci termine une première approche des séquences abstraites. La représentation des séquences abstraites que l'on utilise effectivement dans l'analyseur sera présentée de manière un peu plus formelle dans la partie consacrée à l'algorithme abstrait, et de façon formelle dans la partie définissant le domaine abstrait.

6.2.1.3 La substitution abstraite plus raffinée que β_{in}

On va utiliser une substitution abstraite supplémentaire qui joue le rôle suivant.

La substitution abstraite β_{in} représente un ensemble de substitutions concrètes qui pourront être utilisées pour exécuter une clause donnée. A partir de ces substitutions concrètes, l'exécution de la clause peut réussir ou échouer. Si elle échoue, la substitution concrète en entrée est telle qu'on est sûr de ne jamais atteindre certains atomes de cette clause.

Si on exécute, par exemple, la clause p(X,Y,Z):-X=a,Y=f(b),q(X,Y,Z),W=Z avec la substitution {X/a, Y/s} en entrée, il est certain qu'on ne parviendra jamais à exécuter les littéraux q(X,Y,Z) et W=Z puisque le terme s associé à Y ne s'unifie pas avec f(b).

C'est grâce à une substitution abstraite plus raffinée que β_{in} que l'on va conserver cette information. On suppose qu'on exécute une clause c composée des

littéraux $L_1,...,L_n$. On a vu que la méthode de calcul retenue associe à chacun de ses littéraux L_{i-1} une séquence abstraite B_{i-1}. Chacune de ces séquences abstraites va contenir une substitution plus raffinée que β_{in} qui décrit les substitutions concrètes en entrée de c telles que l'exécution de c se déroule au moins une fois avec succès jusqu'au littéral L_i.

Si l'exécution d'une clause se termine, les propriétés des substitutions résultats satisferont celles décrites dans les substitutions plus raffinées que β_{in} de chacune des séquences abstraites de la clause. L'inverse est aussi vrai : si on utilise une substitution concrète en entrée pour l'exécution de la clause qui satisfait les propriétés décrites dans la substitution plus raffinée que β_{in} de la clause, alors on sait que cette exécution se terminera au moins une fois.

6.2.1.4 Le comportement des procédures

Comme on l'a vu, le système GAIA a pour caractéristique d'effectuer systématiquement tous les calculs nécessaires pour interpréter une procédure donnée. On a également vu que l'utilisation d'une méthodologie de développement est nécessaire pour utiliser au mieux la programmation logique. Dans la méthodologie de Y. Deville, l'utilisateur a la possibilité de donner des informations qui peuvent être utilisées de manière automatique pendant l'analyse d'un programme. Ces informations prennent la forme de directionnalités, de relations entre les tailles des termes entre eux et de relations entre la taille des termes et le nombre de solutions, que ce soit dans une même substitution abstraite ou entre substitutions abstraites d'une même séquence abstraite. Grâce à ces données, il peut intervenir dans l'analyse qui est effectuée : si une des informations données par l'utilisateur n'est pas vérifiée pendant l'analyse, celle-ci se termine. Ces informations vont être regroupées sous le terme de « comportement » de la procédure.

6.2.2 Conséquences de ces nouvelles fonctionnalités

Les conséquences des extensions apportées à GAIA sont multiples, et certaines majeures. Tout d'abord, nous avons vu que l'utilisateur joue une part beaucoup plus active dans le déroulement de l'analyse puisqu'il peut influencer l'évolution et le résultat de celle-ci en décrivant les comportements des procédures comme il l'entend. L'analyseur va vérifier si tous les comportements définis sont bien vérifiés par toutes les procédures, et il ne va rien faire d'autre. Il est donc orienté vers *la vérification de programmes*, alors que GAIA était orienté vers une automatisation complète des calculs. Les résultats de cette analyse pourront, eux servir de base à d'autres opérations (comme l'optimisation). Il peut donc s'intégrer dans un environnement de développement comme étant un outil de préparation à d'autres tâches portant sur les programmes Prolog.

D'autre part, le domaine abstrait de cet analyseur réutilise celui défini dans GAIA pour ce qui est des modes, formes, partages de variables et partages de valeurs. On réutilise aussi (après quelques adaptations) les substitutions abstraites définies dans GAIA. On y a ajouté de nouveaux types d'information tels que les types des termes, les

relations entre les tailles des termes à l'intérieur d'une même substitution abstraite, les relations entre les tailles des termes en entrée et en sortie à l'intérieur d'une même séquence abstraite et les relations entre les tailles des termes et le nombre de solution à l'intérieur d'une même séquence abstraite. Cette notion de séquence abstraite est importante et constitue la grande nouveauté de cet analyseur. Les relations concernant les tailles sont représentées à l'aide d'une bibliothèque C [10] qui prend en charge la gestion de systèmes d'(in)équations et de polyèdres. *Ce nouveau domaine abstrait a la particularité de faire interagir les différents types de données qu'il définit, ce qui améliore la précision des résultats.*

Une autre particularité majeure de cet analyseur est la possibilité donnée au programmeur de définir le comportement des procédures d'un programme. Il a fallu pour cela créer un langage de spécification de comportements. Ce langage, dont la syntaxe peut être trouvée en annexe, lui permet d'identifier les procédures d'un programme et d'y associer les propriétés des substitutions concrètes en entrée utilisables pour exécuter cette procédure, ainsi que les propriétés des substitutions concrètes en entrée qui doivent mener au moins à un succès, et les propriétés des substitutions qui seront retournées par une exécution réussie de celle-ci. Il peut aussi définir des relations entre les tailles des termes en cas de succès et les relations entre les tailles des termes et le nombre de solutions de cette procédure. Les propriétés de substitutions qu'il peut donner sont celles définies dans le domaine abstrait.

Enfin, tel que nous l'avons vu, l'algorithme utilisé pour calculer les séquences abstraites ignore complètement le type d'information contenu dans celles-ci. Il constitue donc un algorithme générique. De plus, les comportements étant disponibles dès le début de l'analyse, l'exécution d'un appel (récursif) ne nécessite plus l'interprétation complète de la procédure appelée avant de connaître le résultat de cette appel. Seule une recherche dans les comportements est réalisée, et les propriétés des substitutions en sortie décrites dans ces comportements correspondent aux résultats de cet appel de procédure. Plus exactement, on recherche le comportement qui décrit le mieux l'appel qui a lieu. Les propriétés des substitutions en sortie décrites dans ce comportement sont alors considérées comme résultats de l'appel[20]. Il n'est donc plus nécessaire de calculer le point fixe d'une procédure lors de l'exécution d'un appel puisque ce point fixe avait pour but de compléter l'information pour une procédure si cette information manquait à un moment donné. Or, ici, elle est présente dès le début et elle est définie pour toutes les procédures. Pour ces raisons, nous changeons d'algorithme générique.

6.2.3 Nouvel algorithme abstrait générique

L'algorithme générique qui va être utilisé, et que nous décrivons un peu plus loin, va se baser sur le calcul des séquences abstraites décrit ci-dessus. Cet algorithme a plusieurs caractéristiques.

---

[20] Nous verrons plus loin qu'il est possible d'approximer au mieux l'exécution de ce type d'instruction en complétant les comportements définis par l'utilisateur par toutes les combinaisons possibles de ces comportements.

6.2.3.1 Une seule analyse

L'analyseur bénéficie d'un plus grand nombre d'informations définissables à l'aide du domaine. L'algorithme va faire interagir ces informations pour obtenir les résultats les plus précis. Il ne s'agit donc pas d'effectuer une série d'analyses, chacune traitant un type d'information particulier comme les modes, les formes et les partages de valeurs, et d'ignorer les relations qui peuvent exister entre les résultats de chacune de ces analyses. L'algorithme que l'on utilise réalise une seule analyse pendant laquelle il va capturer les différents types d'information tout en recherchant la signification de ces informations quand elles sont mises en corrélation.

6.2.3.2 Analyse de programmes Prolog normalisés

Tout comme GAIA, les programmes logiques considérés par l'analyseur sont rédigés en Prolog. Mais un programme Prolog peut très vite avoir une structure très complexe. Cependant, il est toujours possible de traduire un programme logique en un programme logique normalisé. La définition d'un programme normalisé est donnée en annexe.

> *Par exemple, les têtes de clauses peuvent contenir des variables, atomes simples et atomes composés, le corps de la clause peut être composé d'atome foncteur composé d'un nombre quelconque d'atomes composants, eux-mêmes composés, etc.*

Tout comme pour GAIA, l'utilisation d'un programme normalisé simplifie grandement la définition des spécifications et de l'algorithme. Elle simplifie aussi l'implantation de l'analyseur. Tout d'abord, on manipule deux cas simples d'unification : l'unification de deux variables et l'unification d'une variable à un foncteur. De plus, les prédicats et les fonctions n'ont que des variables pour paramètres, jamais des termes plus complexes. Enfin, les substitutions en entrée et en sortie pour une procédure p/n sont toujours exprimées en termes de variables $X_1,...X_n$, ce qui simplifie fortement les renominations de variables (les indices des variables jouant un rôle primordial dans la spécification de l'algorithme).

6.2.3.3 Les résultats de l'analyse

L'analyseur a pour but de vérifier un programme Prolog, c'est à dire de contrôler si les diverses exécutions possibles de ce programme se comportent bien de la façon dont on le suppose. Pour cela, l'analyseur va exécuter chaque clause de chaque procédure à partir des substitutions abstraites en entrée définies dans chacun des comportements de la procédure. S'il parvient à exécuter chacune de ces clauses, et que les résultats retournés par celles-ci sont compatibles avec les substitutions abstraites de sorties définies dans les comportements, alors nous dirons que *l'analyse est un succès*. Si une des clauses ne se termine pas, ou si une des clauses se termine mais retourne une substitution abstraite qui n'a pas les propriétés définies dans le comportement correspondant, nous dirons que *l'analyse est un échec*.

Il est important de dire que l'analyseur ne tire aucune conclusion une fois qu'il a interprété un programme. L'analyse a pour seule fonction de vérifier si un programme donné répond oui ou non aux informations fournies par l'utilisateur[21]. Mais cela ne signifie pas que ce programme soit incorrect si l'analyse échoue. Ce n'est en tout cas pas l'analyseur qui va tirer cette conclusion. Par exemple, si l'analyse détecte qu'une procédure ne génère pas *que* les substitutions de sortie définies par les directionnalités de cette procédure, cela ne signifie pas que la procédure soit incorrecte mais seulement « qu'elle ne génère pas que les substitutions de sortie décrites dans les directionnalités ». En effet, cette procédure peut très bien être utilisée en dehors des conditions définies par les directionnalités et cependant avoir le comportement attendu.

De même, si l'utilisateur a juste besoin de savoir si, dans un cas donné, le programme se comporte comme il le suppose, il analysera celui-ci avec la description du cas en question. Si l'analyse échoue, c'est que ce cas n'est pas vérifié par la procédure, et l'analyseur ne tire aucune conclusion de cet échec. Par contre, si l'analyse est un succès, cela veut seulement dire que le programme se comporte de la façon décrite par l'utilisateur, mais cela ne signifie pas que le programme fait exactement ce qu'on en attend : il peut très bien avoir un comportement compatible avec celui spécifié par l'utilisateur et ne pas répondre du tout à sa spécification.

Cependant, dans le cadre de la méthodologie de Y. Deville, on décrit dans un premier temps une description logique pour chaque procédure. Si cette description est correcte d'un point de vue déclaratif et que l'on associe à celle-ci un ensemble de comportements, on peut estimer que la réussite de l'analyse est une condition suffisante vérifiant la correction du programme.

On le voit, c'est le programmeur qui choisit la manière dont il utilise l'analyseur. On peut aussi décrire des comportements de procédures qui auront pour seul but d'être utilisés lors de l'analyse pour capturer suffisamment d'informations utilisables pendant une optimisation.

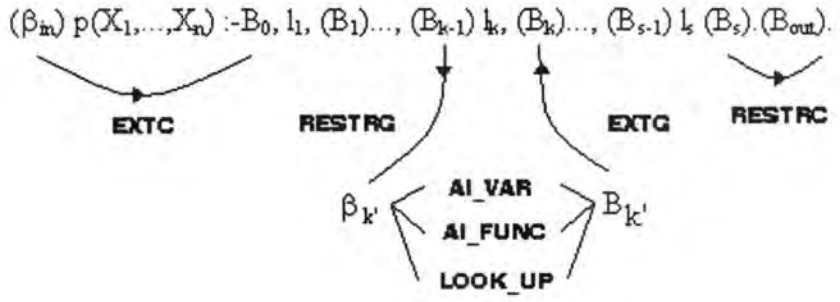
6.2.3.4 Fonctionnement informel de l'analyseur

Nous avons déjà eu un aperçu de la méthode utilisée pour calculer une série de séquences abstraites à partir d'une substitution abstraite en entrée au niveau de la clause. Nous représentons ici cette méthode de manière plus détaillée. Nous verrons aussi comment l'analyseur réalise l'analyse des procédures et du programme.

[21] Et seulement ces informations là.

6.2.3.4.1 Analyse des clauses

L'analyse d'une clause est schématisé par :



La substitution abstraite en entrée β_{in} est toujours définie sur les variables $X_1,...,X_n$ présentes dans la tête de la clause à exécuter. Elle contient les propriétés de chacune de ces variables au début de l'exécution. La première opération à réaliser est l'extension de β_{in} aux variables $X_{n+1},...,X_{n+m}$ présentes dans la clause mais qui n'appartiennent pas à la tête de cette clause. On obtient la substitution abstraite β_{in}'. L'utilisation de programmes normalisés nous facilite déjà la tâche. En effet, l'extension consiste simplement à ajouter dans β_{in} les informations les plus générales sur des variables que l'on sait absentes de β_{in}. D'autre part, β_{in}' est utilisée pour créer une première séquence abstraite $B_0=\langle\beta_{in},\langle\beta_{in}'\rangle\rangle$. Cette séquence est créée pour uniformiser l'exécution du premier littéral à celle des littéraux suivants. Ces deux opérations sont réalisées par l'opération EXTC (extension to clause variables). A partir de là, les exécutions des littéraux se succèdent.

On suppose que l'interprétation des littéraux $l_1,...,l_{k-1}$ a été réalisée et que le résultat de celle-ci est la séquence abstraite B_{k-1}. Cette séquence nous donne les propriétés de toutes les variables de la clause suite à l'exécution de tous ces littéraux. On veut à présent exécuter le littéral l_k. L'utilisation de programmes normalisés va faciliter cette exécution de manière significative.

En effet, un programme normalisé ne réalise que des appels de procédure $q(X_{i_1},...,X_{i_m})$, des unifications de variables $X_{i_1}=X_{i_2}$ et des unifications de variables et de foncteurs $X_{i_1}=f(X_{i_2},...,X_{i_m})$, ce qui réduit à trois le nombre de cas à spécifier. Donc, le littéral l_k que l'on veut exécuter a une de ces trois formes. De plus, les seuls paramètres utilisables par ces littéraux sont les variables X_i. Ces variables ont toutes la même forme et on peut donc facilement les identifier par leur indice i. Les exécutions des unifications de variables entre elles, des unifications de variables et de foncteurs et des appels de procédure sont réalisées respectivement par les opérations AI_VAR (abstract interpretation of variables unification), AI_FUNC (abstract interpretation of a variable and a functor unification), et LOOK_UP (recherche d'un comportement). Chacune de ces opérations a pour substitution abstraite en entrée qui définit les propriétés des résultats de l'exécution des littéraux $l_1,..,l_{k-1}$. L'exécution de ces opérations est

décomposée en trois étapes et retourne une séquence abstraite qui représente les propriétés des termes suite à l'exécution des littéraux $l_1,...,l_k$.

L'exécution d'un littéral se décompose en 3 étapes.

La spécification de l'exécution des littéraux peut être réalisée de manière standard sur une substitution abstraite définie seulement sur les variables utilisées par un littéral donnée. Ainsi, l'unification $X_{i_1}=X_{i_2}$ se fait toujours dans une substitution abstraite définie seulement sur X_{i_1} et X_{i_2}, l'unification $X_{i_1}=f(X_{i_2},...,X_{i_m})$ et l'appel de procédure $q(X_{i_1},...,X_{i_m})$ se feront toujours sur une substitution définie sur $X_{i_1},...X_{i_m}$. S'il y a des variables dans la substitution abstrait initiale qui ne sont pas utilisées pendant cette exécution, il est toujours possible de réaliser dans un premier temps l'opération sans tenir compte de ces variables, et ensuite de propager les effets de l'exécution sur celles-ci. Ainsi, l'opération RESTRG (restriction to goal variables) ne retient d'une substitution abstraite que les variables présentes dans l'instruction qui va être exécutée. L'opération EXTG (extension of goal variables) réalise l'opération inverse : après l'exécution d'un littéral, il est nécessaire de reconsidérer toutes les variables de la clause. Pour cela, EXTG réintègre toutes les variables qui n'ont pas été utilisées par le littéral dans la séquence calculée par l'exécution de ce littéral[22], et propage les effets de cette exécution sur ces variables en tenant compte des relations qui existaient entre toutes les variables avant l'exécution du littéral.

Le seul inconvénient de cette pratique est que l'ensemble des variables X_{i_j} utilisées dans un littéral peut être à chaque fois différent. Or, comme nous l'avons dit, l'utilisation des variables X_{i_j} a pour avantage une identification *facile* des termes par l'index i_j. Pour faciliter cette identification, on décide que l'opération RESTRG va, en plus, renommer les variables $X_{i_1},...X_{i_m}$ en $X_1,...,X_m$. Ainsi, l'opération AI_VAR s'exécutera *toujours* sur les variables X_1 et X_2, et les opérations AI_FUNC et LOOKUP sur les variables $X_1,...,X_m$. De même, l'opération EXTG va en plus rétablir la correspondance entre les variables $X_1,...,X_m$ utilisées pendant l'exécution de l_k avec les variables $X_{i_1},...,X_{i_m}$ utilisées dans la séquence résultat de $l_1,...,l_{k-1}$.

Attention donc que les variables X_i utilisées *pendant* l'exécution d'un littéral ne sont pas équivalentes aux variables X_j utilisées dans les séquences $B_1,...,B_s$ seulement si elles sont identifiées sur base de leurs indices.

L'opération EXTG prendra aussi en charge le calcul de la substitution plus raffinée que β_{in} présente dans chaque séquence abstraite. Ces substitutions, définies dans les B_i, ne peuvent être calculée *qu'après* l'exécution de chaque littéral puisque l'on se base sur les résultats calculés par l'exécution d'un littéral pour rechercher l'ensemble des termes qui, s'il sont utilisés comme paramètres lors de l'exécution de ce littéral, rendront son exécution possible au moins une fois.

Enfin, une fois le dernier littéral l_s exécuté, la séquence abstraite retournée par cette exécution est restreinte aux variables $X_1,...,X_n$ présentes dans la tête de la

---

[22] Séquence définie uniquement sur les variables utilisées dans le littéral.

clause. Cette projection est réalisée par l'opération RESTRC (restriction to clause head variables).

Dans le cas où une unification ne peut pas réussir dans une substitution abstraite donnée, ou bien si un appel de procédure ne se termine pas, on reporte la nom terminaison de la clause dans la séquence abstraite résultant de cet unification ou de cet appel. Toutes les séquences suivantes seront elles aussi marquées comme ne retournant pas de résultat. Au final, la séquence abstraite retournée par la clause indique qu'aucune réponse n'est calculée.

Les opérations vues ci-dessus sont propres à l'algorithme abstrait et opèrent sur des séquences abstraites tout en ignorant la structure de celles-ci. Mais ces opérations vont utiliser d'autres opérations abstraites qui doivent, elles, être définies sur le domaine abstrait et qui doivent tenir compte de la composition des séquences : elles savent qu'elles opèrent sur des substitutions abstraites en entrée, en sortie et plus raffinée que β_{in}, et que deux systèmes d'équations représentent les relations sur les tailles et le nombre de solutions. Elles doivent également tenir compte des informations contenues dans une substitution abstraite pour réaliser une analyse qui met en correspondance les différents types d'informations afin d'affiner les résultats. On va donc réutiliser les opérations définies dans GAIA, mais on doit aussi redéfinir ces opérations pour implanter ces correspondances. Toutes ces redéfinitions seront spécifiées dans les chapitres suivants.

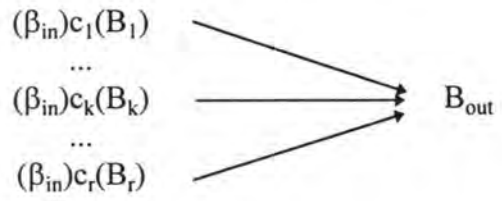
6.2.3.4.2 Analyse des procédures

L'analyse d'une procédure consiste en l'analyse de chacune des clauses et en une représentation unique des résultats retournés par ces clauses.

Pour un comportement donné, on va analyser chaque clause de la procédure à interpréter à partir de la substitution en entrée définie dans ce comportement. On calcule ainsi l'ensemble des séquences abstraites retournées par chaque clause dans un cas d'utilisation donné. Une fois toutes les séquences résultats retournés[23], on crée une séquence abstraite B_{out} qui représente à elle seule toutes les propriétés décrites par les différentes séquences abstraites reçues. Cette séquence abstraite représente donc toutes les propriétés calculées sur l'ensemble des clauses composant la procédure à partir d'une substitution en entrée donnée. Une fois B_{out} connue, on compare les propriétés qu'elle décrit à celles définies par l'utilisateur dans le comportement. Si une seule des spécifications de l'utilisateur n'est pas vérifiée par B_{out}, on dit que *l'analyse de la procédure est un échec*, sinon on dit que *l'analyse de la procédure est un succès*.

[23] Chacune représentant donc un ensemble de propriétés sur les variables quand on exécute une clause avec une substitution abstraite en entrée spécifique.

Si la procédure est composée des clauses $c_1,...,c_r$ et que la substitution abstraite en entrée est β_{in}, on peut schématiser l'analyse de la procédure par :



6.2.3.4.3 Analyse des programmes

L'analyse d'un programme va effectuer l'analyse de chacune des procédures du programme à partir des substitutions abstraites en entrée définies dans chacun des comportements associés aux procédures. Si toutes les analyses de procédures réalisées sont des succès, alors *l'analyse du programme est un succès*, sinon, c'est *un échec*.

6.2.3.5 Exemple : Select/3

Nous allons illustrer cette présentation à l'aide de la procédure select/3. Les propriétés qui vont être capturées sur cette procédures sont les modes, les types, les formes et les partages de variables et de valeurs. Comme le domaine abstrait de l'analyseur n'a pas encore été défini, les séquences et substitutions abstraites utilisées ici sont encore très littéraires. De même pour le comportement. De plus, pour ne pas surcharger les résultats, seules les informations capturées y sont représentées.

Nous utiliserons la notation t_i pour identifier un terme associé à une variable X_j et qui identifie les propriétés de cette variable. Cette notation est le moyen choisi pour représenter le partage de valeurs entre deux variables : si elles sont associées au même terme t_i, alors elle partage la valeur de ce terme. Cependant, ces t_i sont propres à la substitution abstraite dans laquelle ils sont définis. L'indice i n'est donc qu'un moyen d'identifier le terme dans une seule substitution. Ils n'identifient pas le même terme dans des substitutions différentes.

La procédure select/3 est :

select(X,L,LS) :- L=[H|T], X=H, LS=T, list(T).
select(X,L,LS) :- L=[H|T], LS=[H|TS], select(X,T,TS).

Supposons que la procédure select/3 normalisée générée par l'analyseur est :

select(X0 , X1 , X2) :- X3 = [X4 | X5] , X1 = X3, X4 = X0, X2 = X5, liste(X5).
select(X0 , X1 , X2) :- X3 = [X4 | X5] , X1 = X3, X6 = [X4 | X7] ,
X2 = X6, select(X0, X5, X7).

Le comportement que l'on considère est :

Si la substitution en entrée qui est utilisée respecte les propriétés suivantes :

{X0/var, X1/liste close, X2/var} et où X0, X2 sont des variables distinctes

alors la substitution plus raffinée que β_{in} et la substitution de sortie qui seront calculées auront respectivement les propriétés suivantes :

{X0/var, X1/[t_1 clos | t2 liste close] liste close, X2/var} et où X0, X2 sont des variables distinctes pour la substitution de raffinement et

{X0/clos, X1/[t_1 clos | t_2 liste close] liste close, X2/clos} pour la substitution de sortie.

6.2.3.5.1 Exécution de la première clause

Extension de la substitution en entrée.

L'exécution de la première clause se déroule comme suit :

(β_{in_1}) select(X0 , X1 , X2) :- **(B_0)** X3 = [X4 | X5] , **(B_1)** X1 = X3,
(B_2) X4 = X0, **(B_3)** X2 = X5,
(B_4) liste(X5) **(B_5)**. **(B_{out_1})**

La substitution en entrée pour l'exécution de la première clause est :

β_{in_1}={X0/t_0 var, X1/t_1 liste close, X2/t_2 var} où X0, X2 sont des variables distinctes

Cette première clause est étendue aux variables de la clause non présentes dans la tête, c'est à dire X3, X4 et X5. Le résultat est la séquence abstraite B_0 :

B_0=⟨ β_{in_1}', β_{ref_0}, β_{out_0}⟩ où :
β_{in_1}' ={X0/t_0 var, X1/t_1 liste close, X2/t_2 var} et où X0, X2 sont distinctes.
β_{ref_0} =β_{in}'
*β_{out_0} ={X0/t_0 var, X1/t_1 liste close, X2/t_2 var, **X3/t_3 var, X4/t_4 var, X5/t_5 var**} et où X0, X2, **X3, X4, X5** sont distinctes.*

Par convention, les changements apportés dans les différentes séquences sont notés en gras.

On voit que, ne sachant rien des nouvelles variables introduites dans la substitution β_{in}, leurs propriétés sont les plus générales possibles : dans notre cas, on ne peut dire qu'une chose, c'est que ce sont des variables distinctes. Cette séquence B_0 est la base de tous les calculs qui auront lieu par la suite.

Exécution du premier atome.

L'exécution de l'atome X3=[X4|X5] se déroule à partir de la substitution β_{out_0}. On commence par projeter les variables utilisées dans l'atome (X3, X4 et X5) hors de cette substitution, et on renomme ces variables en X0, X1, X2. On obtient la substitution intermédiaire β_{inter_0}. Ces variables *ne sont pas* équivalentes aux variables *de même indice* dans B_0.

β_{inter_1}={X0/t_0 var, X1/t_1 var, X2/t_2 var} où les variables X0, X1 et X2 sont distinctes.

L'unification va déterminer la forme du terme associé à la variable X0. Parce qu'elle est associée à un terme composé, il faut modifier l'information sur X0 contenue dans β_{inter_1}. Après unification X0 est associé à un terme unifiable à une liste et composé des termes associés à X1 et X2. L'unification va aussi nous permettre de construire la substitution plus raffinée que β_{in} définies sur les variables de l'atome et telle que, si les termes associés aux variables du littéral ont les propriétés définies par cette substitution plus raffinée que β_{in} au moment de l'exécution de l'atome, l'exécution de cette atome réussit au moins une fois (dans ce cas, il réussit exactement une fois). Cette substitution β_{ref_1}, est calculée sur base de β_{out_1}, et de β_{inter_1}. Pour cette opération, aucune nouvelles propriétés (par rapport à β_{inter_1}) n'est calculée, mais ce ne sera plus le cas par la suite.

L'unification retourne la séquence intermédiaire B_{aux_1} :

B_{aux_1}=⟨β_{inter_1}, β_{ref_1}', β_{out_1}'⟩ où :
β_{ref_1}' = {X0/t_0 var, X1/t_1 var, X2/t_2 var} où les variables X0, X1 et X2 sont distinctes.
*β_{out_1}' = {X0/**t_0 non clos de forme [t_1 \| t_2]**, X1/t_1 var, X2/t_2 var} où **X1 et X2 sont distinctes.***

B_{aux_1} est ensuite étendue aux variables présente dans β_{out_0} et qui ne sont pas utilisées dans l'instruction qui vient d'être exécutée. Il faut à ce niveau bien faire la distinction entre les indices utilisés dans B_{aux_1} et dans β_{out_0} : il s'agit d'appliquer sur les indices de B_{aux_1} la correspondance qui a été établie par RESTRG, afin de retrouver pour chaque terme de B_{aux_1} l'indice correspondant dans β_{out_0}. De plus, il faudra propager les effets de l'unification sur l'ensemble des termes qui sont liés aux variables utilisées par le littéral. Par la suite, nous désignerons cette opération pas « extension et propagation » et nous la compléterons si nécessaire.

Tous ces résultats sont dans la substitution B_1.

B_1=⟨β_{in_1}', β_{ref_1}, β_{out_1}⟩
β_{ref_1}=β_{in}'
*β_{out_1}={X0/t_0 var, X1/t_1 liste close, X2/t_2 var, **X3/t_3 non clos de forme [t_4\|t_5]**, X4/t_4 var, X5/t_5 var} avec **X0,X2,X4 et X5 distinctes**.*

Cette séquence nous donne les propriétés des résultats de la clause après l'exécution du premier atome. On voit qu'on dispose d'une information plus détaillée sur les termes qui peuvent être associés à X3. On voit aussi B_1 a la même substitution en entrée que B_0. Il en sera de même pour toutes les séquences qui seront calculées.

La substitution β_{out_1} nous indique un partage de valeur entre X3, X4 et X5. Tout ce qui surviendra par la suite sur les propriétés des termes associés à ces variables aura une conséquence sur toutes les variables qui partage ces propriétés.

Exécution du second atome.

L'exécution de l'atome X1=X3 se fait à partir de β_{out_1}. Encore une fois, on projette les variables X1et X3 de cette substitution et on les renomme respectivement en X0 et X1. Or, le terme associé à X3 est composé des termes associés aux variables X4 et X5. On veut également projeter cette information, mais sans projection des variables X4 et X5 qui n'apparaissent pas dans le second atome. Ces termes sont renommés en t_2 et t_3 respectivement. On crée ainsi la substitution temporaire β_{inter_2}.

$\beta inter_2$={X0/t_0 liste close, X1/t_1 non clos de la forme [t_2 var|t_3 var]} avec les termes t_2 et t_3 distincts.

De nouveau, l'unification donne au terme associé à X0 la même forme que le termes associé à X1. Mais nous avons un exemple d'interaction possible entre les différents types d'information. On savait au départ que X0 était une liste close mais on ignorait quels étaient les termes qui le composaient. On savait également que X1 était associé à un terme unifiable à une liste et composé de deux variables. L'unification va nous permettre d'affirmer que X0 est donc associé à une liste composée des termes t_2 et t_3 *et que* t_2 est un terme clos quelconque *et que* t_3 est une liste close, *et par conséquent*, X1 est associé à une liste close. t_2 et t_3 étant des variables au départ distinctes, leurs unifications se font indépendamment et sans conditions particulières.

D'autre part, il est possible d'affiner la substitution initiale β_{inter_2} grâce à la substitution β_{out_2}, résultant de l'unification du littéral. Comme on le voit dans β_{ref_2}, ci-dessous, si la forme des termes associés à X0 est composée d'un terme clos et d'une liste close, l'exécution de l'atome est toujours un succès. Cette information peut, à ce moment-ci, sembler redondante puisque toute liste close associée à X0 est toujours composée d'un terme clos et d'une liste close. Par contre, elle sera utile dans la séquence B_2 que nous verrons plus loin, car elle permettra d'affirmer que, si le terme associé à X1 est une liste close, alors l'exécution de la clause réussira jusqu'au deuxième atome au moins.

L'unification retourne la séquence B_{aux_2} suivante :

$B aux_2$=($\beta inter_2$, $\beta ref'_2$, βout_2') où :
*βref_2'= **{X0/t_0 liste close de la forme [t_4 clos | t_5 liste close]**, X1/t_1 non clos de la forme [t_2 var | t_3 var]} avec les termes t_1 et t_2 distincts.*
*βout_2'=**{X0/t_0 liste close de la forme [t_1 clos | t_2 liste close], X1/t_0}**.*

B_{aux_2} est ensuite étendue aux variables présente dans β_{out_1} et qui ne sont pas dans l'instruction qui vient d'être exécutée. On voit ici que seules les variables X1 et X3 avaient été projetées, mais comme le terme associé à X3 est composé des variables X4 et X5, on a que ces variables sont elles aussi touchées par l'unification qui vient d'avoir

lieu. Il ne faut donc pas seulement remplacer les termes associés aux variables X1 et X3 par leur nouvelles valeurs pour obtenir la séquence résultant de l'exécution du littéral : il faut aussi propager les effets de l'unification de ces variables sur l'ensemble des variables qui sont liées à X1 et X3. Tout ceci est pris en charge par l'opération EXTG sur B_{aux_1}. Cette opération retourne la séquence B_2 :

$B_2 = (\beta in_1', \beta ref_2, \beta out_2)$ où :
*βref_2={X0/t_0 var, **X1/t_1 liste close de la forme [t_3 clos \| t_4 liste close]**, X2/t_2 var}*
avec X1 et X2 variables distinctes
*βout_2={X0/t_0 var , X1/t_1 liste close de la forme **[t_4\|t_5]**, X2/t_2 var, **X3/t_1** , **X4/t_4** **clos** , **X5/t_5 liste close**} avec **X0 et X2 distinctes**.*

On le voit, les effets de l'unification de X1 et X3 a eu des répercussions sur les termes liés à X4 et X5.

Exécution du troisième atome.

On exécute de X4=X0 à partir de la substitution β_{out_2}. On projette les variables X4 et X0 et on les renomme respectivement X0 et X1. On obtient la substitution d'entrée pour l'exécution de l'atome.

$\beta inter_3$ = {X0/t_0 clos, X1/t_1 var}

Le raisonnement de l'unification ayant déjà été expliqué ci-dessus, on passe tout de suite au résultat. L'unification retourne la séquence B_{aux_3} définie sur les variables de l'atome.

$B aux_3 = (\beta inter_3, \beta ref_3', \beta out_3')$ où :
$\beta ref_3'$={X0/t_0 clos, X1/t_1 var}
*$\beta out_3'$={X0/t_0 clos, **X1/t_0** }*

On effectue l'extension et la propagation des résultats à partir de B_{aux_3}. On obtient la séquence B_3 résultant de l'exécution de la clause jusqu'au troisième atome.

$B_3 = (\beta in_1', \beta ref_3, \beta out_3)$ où :
βref_3={X0/t_0 var, X1/t_1 liste close de la forme [t_3 clos \| t_4 liste close], X2/t_2 var} où X0 et X2 sont distinctes ;
*βout_3={X0/t_0 **clos**, X1/t_1 liste close de la forme **[t_0\|t_5]**, X2/t_2 var, X3/t_1, **X4/t_0**, X5/t_5 liste close}.*

Exécution du quatrième atome.

L'exécution de X2=X5 se fait à partir de β_{out_3}. On obtient β_{inter_4} où X2 est renommée X0 et X5 est renommée X1.

$\beta inter_4$={X0/t_0 var, X1/t_1 liste close}

L'unification retourne la séquence B_{aux_4}.

$B_{aux_4}=\langle \beta_{inter_4}, \beta_{ref_4}', \beta_{out_4}' \rangle$ où :
$\beta_{ref_4}'=\{X0/t_0$ *var*, $X1/t_1$ *liste close*$\}$
$\beta_{out_4}'=\{X0/t_0$ ***liste close***, $\mathbf{X1/t_0}\}$

Il fait de nouveau propager les résultats de l'unification sur les termes liés à X2 et X5 dans β_{out_3}. Dans notre cas, X2 et X5 représentant maintenant le même terme, il est possible de « supprimer » une des deux variables pour ne garder (par exemple) que celle de plus petit indice. Ainsi, la structure de X1 est modifiée et X5 est associé à X1(constitue un renvoi vers X1).

$B_4=\langle \beta_{in_1}', \beta_{ref_4}, \beta_{out_4} \rangle$ où :
$\beta_{ref_4}=\{X0/t_0$ *var*, $X1/t_1$ *liste close de la forme* $[t_3$ *clos* $\mid t_4$ *liste close*$]$, $X2/t_2$ *var*$\}$ *avec X0 et X2 distinctes ;*
$\beta_{out_4}=\{X0/t_0$ *clos*, $X1/t_1$ *liste close de la forme* $[t_0|\mathbf{t_2}]$, $X2/\mathbf{t_2}$ ***liste close***, $X3/t_1$, $X4/t_0$, $X5/\mathbf{t_2}\}$.

Exécution du cinquième atome.

L'exécution de l'atome liste(X5) va utiliser le comportement défini pour le prédicat liste(X0). On suppose que ce comportement est le suivant :

Si la substitution en entrée qui est utilisée respecte les propriétés suivantes :

$\{X0/t_0$ *liste close*$\}$

alors la substitution plus raffinée que celle en entrée et la substitution de sortie qui seront calculées auront respectivement les propriétés suivantes :

$\{X0/t_0$ *liste close*$\}$ *pour la substitution de raffinement et*
$\{X0/t_0$ *liste close*$\}$ *pour la substitution de sortie.*

L'exécution de liste(X5) se fait à partir de β_{out_4}. On projette X5 et on la renomme X0. On obtient la substitution initiale β_{inter_5}.

$\beta_{inter_5}=\{X0/t_0$ *liste close*$\}$

On va rechercher dans les comportements de la procédure liste une substitution en entrée qui définit de la façon la plus précise les propriétés définies dans β_{inter_5}. Dans notre cas, un seul comportement est défini et les informations qu'il définit pour les substitutions plus raffinées que β_{in} et de sortie constitue les résultats de l'exécution de list(X5). A partir de ces informations, on crée la séquence B_{aux_5}.

$B_{aux_5}=\langle \beta_{inter_5}, \beta_{ref_5}', \beta_{out_5}' \rangle$ où :
$\beta_{ref_5}'=\{X0/t_0$ *liste close*$\}$
$\beta_{out_5}'=\{X0/t_0$ *liste close*$\}$

L'extension et la propagation retourne la séquence B_5.

$B_5 = \langle \beta in_1', \beta ref_5, \beta out_5 \rangle$ *où :*

βref_5 *= {X0/*t_0 *var, X1/*t_1 *liste close de la forme [*t_3 *clos | *t_4 *liste close], X2/*t_2 *var} où X0 et X2 distinctes.*

βout_5*={X0/*t_0 *clos, X1/*t_1 *liste close de la forme [*t_0*|*t_2*], X2/*t_2 *liste close, X3/*t_1*, X4/*t_0*, X5/*t_2*}*

Aucune modification n'est apportée à cette séquence, mais l'atome exécuté *a réussi*. On sait donc que la clause s'exécute jusque la fin.

Restriction aux variables de tête.

L'exécution de la clause est terminée et la séquence B_5 décrit les propriétés des termes après cette exécution. On restreint à présent B_5 aux variables X0, X1 et X2 afin de ne retenir que les propriétés des paramètres de la première clause. On obtient la séquence abstraite B_{out_1}, résultat de la première clause :

$B_{out_1} = \langle \beta in_1, \beta ref_{B1}, \beta out_{B1} \rangle$ *où ;*

βref_{B1}*={X0/*t_0 *var, X1/*t_1 *liste close de la forme[*t_3 *clos | *t_4 *liste close], X2/*t_2 *var} avec X0 et X2 distincts*

βout_{B1}*={X0/*t_0 *clos, X1/*t_1 *liste close de la forme [*t_0*|*t_2*], X2/*t_2 *liste close}*

6.2.3.5.2 Analyse de la deuxième clause

La seconde clause va se dérouler comme suit :

(β_{in_2}) select(X0 , X1 , X2) :- **(B_0)** X3 = [X4 | X5] , **(B_1)** X1 = X3, **(B_2)** X6 = [X4 | X7] , **(B_3)** X2 = X6, **(B_4)** select(X0,X5,X7) **(B_5)** . B_{out_2}

La substitution en entrée pour l'exécution de cette clause est :

βin_2*={X0/*t_0 *var, X1/*t_1 *liste close, X2/*t_2 *var} où X0, X2 sont des variables distinctes*

Les quatre premières unifications étant semblables à celles de la première clause, nous ne ferons que donner les séquences B_0, B_1, B_2, B_3 et B_4. Nous verrons ensuite l'exécution de l'appel récursif select.

Séquence abstraite B_0.

$B_0 = \langle \beta in_2', \beta ref_0, \beta out_0 \rangle$ *où :*

$\beta in_2'$ *={X0/*t_0 *var, X1/*t_1 *liste close, X2/*t_2 *var} où X0, X2 sont des variables distinctes*

$\beta ref_0 = \beta in_2'$

βout_0*={X0/*t_0 *var, X1/*t_1 *liste close, X2/*t_2 *var,* ***X3/*t_3 *var, X4/*t_4 *var, X5/*t_5 *var,***

X6/t_6 var, X7/t_7 var} *et X0, X2,* ***X3, X4, X5, X6 et X7*** *sont des variables distinctes.*

Séquence abstraite B_1.

$B_1 = \langle \beta in_2', \beta ref_1, \beta out_1 \rangle$ *où :*

$\beta ref_1 = \beta in_2'$

βout_1=*{X0/t_0 var, X1/t_1 liste close, X2/t_2 var, X3/t_3* ***non clos de la forme [t_4\|t_5]****, X4/t_4 var, X5/t_5 var, X6/t_6 var, X7/t_7 var} et* ***X0, X2, X4, X5, X6, X7 distinctes****.*

Séquence abstraite B_2.

$B_2 = \langle \beta in_2', \beta ref_2, \beta out_2 \rangle$ *où :*

βref_2=*{X0/t_0 var, X1/t_1 liste close de la forme* ***[t_3 clos \| t_4 liste close]****, X2/t_2 var} et X0, X2 distinctes.*

βout_2=*{X0/t_0 var, X1/t_1 liste close de la forme* ***[t_4\|t_5]****, X2/t_2 var, X3/****t_1****, X4/t_4* ***clos****, X5/t_5* ***liste close****, X6/t_6 var, X7/t_7 var} et* ***X0,X4,X5,X6 distinctes****.*

Séquence abstraite B_3.

$B_3 = \langle \beta in_2', \beta ref_3, \beta out_3 \rangle$ *où :*

βref_3=*{X0/t_0 var, X1/t_1 liste close de la forme [t_3 clos \| t_4 liste close], X2/t_2 var} et X0 et X2 distinctes.*

βout_3=*{X0/t_0 var, X1/t_1 liste close de la forme [t_4\|t_5], X2/t_2 var, X3/t_1, X4/t_4 clos, X5/t_5 liste close, X6/t_6* ***non clos de la forme [t_4\|t_7]****, X7/t_7 var} et* ***X0, X7 distinctes****.*

Séquence abstraite B_4.

$B_4 = \langle \beta in_2', \beta ref_4, \beta out_4 \rangle$ *où :*

βref_4=*{X0/t_0 var, X1/t_1 liste close de la forme [t_3 clos \| t_4 liste close], X2/t_2 var} et X0,X2 distinctes.*

βout_4=*{X0/t_0 var, X1/t_1 liste close de la forme [t_4\|t_5], X2/t_2* ***non clos de la forme [t_4\|t_7]****, X3/t_1, X4/t_4 clos, X5/t_5 liste close, X6/****t_2****, X7/t_7 var} avec X0,X7 distinctes.*

Exécution de select(X0,X5,X7).

Cette exécution est similaire à celle de l'atome liste(X5) de la première clause. On commence par projeter les variables X0, X5 et X7 et par les renommer respectivement X0, X_1 et X2. On obtient β_{inter_5}.

$\beta inter_5$=*{X0/t_0 var, X1/t_1 liste close, X2/t_2 var} avec X0 et X2 distinctes.*

On recherche dans les comportements de select un comportement qui défini « au mieux » les propriétés décrites dans β_{inter_5}. On trouve le comportement décrit au début de ce chapitre, et à partir duquel on construit la séquence B_{aux_5}. On considère que les informations décrites dans le comportement donnent les propriétés des termes après

exécution d'un appel à la procédure select. Le fait qu'il s'agisse d'un appel récursif ne nécessite pas que l'on réexécute la procédure select. On doit cependant vérifier que la taille des termes utilisés comme paramètres de l'appel décroît strictement si on veut vérifier la terminaison de l'appel récursif. Or ici, cette condition est vérifiée puisque X5 est associé au suffixe du terme associé à X1.

B_{aux_5}=(β_{inter_5}, β_{ref_5}', β_{out_5}') où :
β_{ref_5}'={X0/t_0 var, X1/t_1 liste close de la forme ***[t_3 clos | t_4 liste close]****, X2/t_2 var} et X0,X2 distinctes.*
β_{out_5}'={X0/t_0 ***clos****, X1/t_1 liste close de la forme* ***[t_3 clos | t_4 liste close]****, X2/t_2* ***liste close}***

L'extension et propagation de B_{aux_5} retourne B_5.

B_5=(β_{in_2}', β_{ref_5}, β_{out_5}) où :
β_{ref_5}={X0/t_0 var, X1/t_1 liste close[t_3 clos| t_4 liste close], X2/t_2 var} et X0,X2 distinctes.
β_{out_5}={X0/t_0 ***clos****, X1/t_1 liste close [t_4|t_5], X2/t_2* ***liste close*** *de la forme [t_4|t_7], X3/t_1, X4/t_4 clos, X5/t_5 liste close de la forme* ***[t_8 clos|t_9 liste close]****, X6/t_2, X7/t_7* ***liste close}***

L'exécution de la clause est terminée. On restreint B_5 aux variables de tête tout en mémorisant les propriétés des variables que l'on supprime qui concernent aussi des variables que l'on retient. On obtient la séquence abstraite résultat de la deuxième clause B_{out_2}.

B_{out_2}=(β_{in_2}, $\beta_{ref_{B2}}$, $\beta_{out\ B2}$) où :
$\beta_{ref\ B2}$= {X0/t_0 var, X1/t_1 liste close de la forme[t_3 clos | t_4 liste close], X2/t_2 var} et X0,X2 distinctes.
$\beta_{out\ B2}$={X0/t_0 clos, X1/t1 liste close de la forme [t_3 clos | t_4 liste close de la forme [t_5 clos | t_6 liste close]], X2/t_2 liste close de la forme[t_3 clos | t_7 liste close]}

6.2.3.5.3 Les résultats de la procédure

Les propriétés des résultats retournés par la procédure sont construits sur base des résultats retournés par chacune des clause. Dans notre cas, les résultats de la procédure sont représentés dans la séquence abstraite $B_{out_{proc}}$. Elle est construite à partir de B_{out_1} et de B_{out_2} et doit représenter toutes les propriétés définies dans chacune des séquences B_{out_1} et de B_{out_2}.

$B_{out_{proc}}$=(β_{in}, $\beta_{ref_{proc}}$, $\beta_{out_{proc}}$) où :
$\beta_{ref_{proc}}$ ={X0/ t_0 var, X1/t_1 liste close de la forme [t_3 clos | t_4 liste close], X2/t_2 var} et X0,X2 distinctes.

βout_{proc}=*{X0/ t_0 clos, X1/t_1 liste close de la forme [t_3 clos |t_4 liste close], X2/t_2 liste close}*

Rappelons que l'analyse de la procédure s'est faite à partir d'une même substitution en entré β_{in} pour les deux clauses. L'analyse de la procédure est donc un succès si les informations calculées et représentées par $\beta_{ref_{proc}}$ et $\beta_{out_{proc}}$ satisfont toutes les propriétés définies dans le comportement en question. Ce qui est le cas dans cet exemple. Si d'autres comportements étaient définis pour cette procédure, il faudrait réexécuter l'analyse de select à partir de la substitution en entrée de chacun d'eux et comparer les résultats calculés à chacune de leurs substitutions correspondantes.

6.2.3.5.4 Les résultats du programme

Le programme étant composé de la seule procédure select, le résultat de l'analyse du programme correspond à celui de l'analyse de la procédure.

7. L'algorithme d'interprétation abstraite générique

Après un bref rappel du fonctionnement de l'analyseur, nous décrivons de manière formelle l'ensemble des opérations qui constituent l'algorithme générique, et nous donnons la spécification de chacune d'elles. Nous donnons aussi la définition des structures propres à l'algorithme, comme les séquences et les substitutions abstraites, l'ensemble des comportements définis par l'utilisateur et la structure « sat ».

Il est important de rappeler que cet algorithme est générique. Il se base donc sur un domaine générique. En d'autres mots, on ignore en ce moment sur quoi *exactement* il va porter. On sait seulement qu'il va opérer sur des séquences abstraites et des substitutions abstraites. Nous rappellerons d'ailleurs ce que ces notions signifient, mais ce n'est pas à ce niveau que nous spécifierons *de quoi* elles sont composées. En effet, l'algorithme utilise les substitutions abstraites pour commencer l'exécution de chaque clause et de chaque atome, et il utilise les séquences pour représenter les résultats de toutes les exécutions. Mais il ignore ce qui se passe précisément dans ces structures au fur et à mesure que l'analyse se déroule.

Par exemple, les propriétés qui seront représentées par ces structures dépend du domaine abstrait choisi. Nous pourrions ainsi avoir des séquences abstraites composées seulement d'une substitution abstrait en entrée et d'une autre en sortie, ou encore composées d'une substitution abstraite en entrée et d'une liste de substitutions abstraites en sortie et complétée par l'ensemble de toutes les relations possibles entre les termes à quel que moment

que ce soit. De même, les substitutions abstraites peuvent représenter l'information sur les modes et les types uniquement, ou bien seulement sur les formes. On le voit, la structure de ces deux notions et les informations qu'elles peuvent représenter est inconnu pour le moment.

C'est pour cette raison que nous présentons d'abord l'algorithme générique : pour montrer que sa spécification ignore la composition des séquences et des substitutions abstraites qui sont utilisées. Nous le verrons, cette spécification ne reprend que les propriétés qui doivent être vérifiées par les séquences et les substitutions abstraites pendant l'exécution de l'analyse, et ces propriétés sont exprimées en termes de séquences concrètes et de substitutions concrètes[1]. Pour cette même raison, les exemples présentés ici se basent sur des substitutions concrètes.

Le chapitre suivant cette présentation générale est consacré à la définition formelle du domaine abstrait choisi. C'est à ce moment que l'on définira la structure et les informations reprises dans les séquences et les substitutions abstraites et qu'il sera possible de donner l'implantation des procédures spécifiées ici. C'est pourquoi une implantation « générale »[2] se trouve en annexe de ce mémoire.

7.1 Structures de données utilisées par l'algorithme générique

Comme nous venons de le voir, l'algorithme se base sur l'utilisation des notions de séquences abstraites et de substitutions abstraites. Bien que ces structures ne seront précisées que plus tard, on donne ici leur définition, afin de pouvoir présenter les opérations de l'algorithme. Cet algorithme se base également sur les comportements définis par l'utilisateur et sur une structure plus complète appelée sat.

7.1.1 Substitutions abstraites

Alors qu'une substitution concrète associe un terme concret à chaque variable du programme, une substitution abstraite associe à chaque variable du programme les propriétés des termes qui peuvent être associés à cette variable dans une substitution concrète. De plus, les substitutions abstraites que l'on va considérer sont capable de donner de l'information sur n'importe quel terme qui compose d'autres termes.

Parce que le domaine abstrait est encore à définir, nous ne pouvons pour le moment donner la sémantique exacte d'une substitution abstraite en se basant sur ses composants. Mais comme il est nécessaire d'en définir une pour spécifier les opérations de l'algorithme abstrait, nous allons nous baser sur les substitutions concrètes. La sémantique d'une substitution abstraite β est donnée par sa fonction de concrétisation (notée $Cc(\beta)$). La concrétisation d'une substitution abstraite génère toutes les substitutions concrètes θ dont les termes satisfont les propriétés définies dans β.

---

[1] C'est à dire, indépendamment de la structure des séquences et des substitutions abstraites que l'on choisira.

[2] Nous donnons une implantation plus proche des notations mathématiques que d'un langage de programmation particulier. L'implantation réelle à quant à elle été réalisée en C++.

7.1.2 Séquences abstraites

Les séquences abstraites ont déjà été introduites dans la présentation générale de l'analyseur. Rappelons que leur but est de capturer de l'information sur le *déroulement d'une exécution* d'une clause ou d'une procédure. En effet, cette exécution peut ne pas retourner de substitutions, retourner une ou plusieurs substitutions, ou encore chercher indéfiniment une substitution.

En regard de ce que nous avons dit dans l'introduction de ce chapitre, c'est tout ce que nous pouvons dire sur les séquences abstraites pour le moment : ne sachant pas de quoi elles seront composées, on ne peut que donner leur rôle. Une séquence abstraite est donc une structure capable de donner de l'information sur le déroulement de la clause et sur l'état d'exécution de cette clause à un moment donné.

La sémantique d'une séquence abstraite est donnée par sa fonction de concrétisation. Encore une fois, nous ne pouvons définir celle-ci de manière précise pour le moment. Cependant, on va considérer dans la suite de ce chapitre[3] que la concrétisation d'une séquence abstrait B (notée Cc(B)) génère l'ensemble de toutes les séquences de substitutions concrètes $\langle\theta_{in},\langle\theta_1,...,\theta_n\rangle\rangle$ qui répondent aux propriétés définies dans B. θ_{in} est la substitution concrète en entrée et $\langle\theta_1,...,\theta_n\rangle$ est la suite des substitutions concrètes qui résultent de l'exécution à partir de θ_{in}. Cette suite peut être infinie, incomplète ($\langle\theta_1,...,\theta_n,\perp\rangle$) ou vide ($\langle\rangle$).

7.1.3 Les comportements de procédures

Le comportement abstrait d'une procédure est une formalisation des spécifications du « comportement de la procédure » fournie par l'utilisateur. Elles lui permettent de définir certaines propriétés que devront satisfaire les procédures tout au long de l'analyse.

- Un comportement abstrait d'une procédure p/n, ou comportement de p, se note Beh_p. C'est un ensemble de tuples $\langle p/n, \{\langle B_1,se_1\rangle,...,\langle B_m,se_m\rangle\}\rangle$ où :
  - $B_1,...,B_m$ sont des séquences abstraites toutes définies sur les variables de tête des clauses composant p/n[4] ;
  - $se_1,...,se_m$ sont des expressions linéaires positives définies sur les variables de tête des clauses composant p/n.

Chaque séquence abstraite B_i définit les classes d'appels possibles et les comportements associés à la procédure p. Se_i est une expression utilisée pour calculer la taille des termes présents dans B_i.

- Sbeh = $(Beh_p)p \in P$ est un ensemble fini de comportements abstraits pour le programme P. Il contient exactement un comportement Beh_p pour chaque procédure p présente dans P. On suppose que l'information donnée par Sbeh est correct pour tous les appels qui peuvent survenir lors de l'exécution de P.

---

[3] Afin de spécifier les opérations de l'algorithme générique.

[4] $dom_{in}(B_k) = dom_{out}(B_k) = \{X_1,...,X_n\}$.

L'utilisateur spécifie les comportements des procédures à l'aide d'un *langage de spécification des comportement*. Ce langage est décrit en annexe. La définition de ce langage est fort liée à la représentation des séquences abstraites définies ici. Ces comportements doivent bien sûr être disponibles avant de commencer l'analyse du programme.

7.1.4 Sat

L'ensemble des comportements définis par l'utilisateur peut suffire à lui seul pour déterminer si les résultats calculés par l'analyseur respectent bien les comportements des procédures. Mais pendant l'exécution d'une clause, il peut y avoir des appels à *d'autres* procédures. Il faut alors que l'analyseur recherche dans les comportements de celles-ci celui qui décrit le mieux l'appel en cours, et qu'il considère le comportement trouvé comme étant le résultat de l'appel. Mais il se peut que ce comportement décrive de manière *beaucoup trop générale* l'appel actuel, et que la description des résultats qu'il donne soit elle aussi beaucoup trop générale. Il se pourrait même que la généralité de ce comportement nuise de manière significative à la précision des résultats de l'interprétation.

La solution est de définir pour chaque procédure du programme un ensemble de tuples abstraits (set of abstract tuples, ou sat) qui va non seulement contenir les comportements spécifiés par l'utilisateur, mais aussi tous les comportements plus précis que l'on peut calculer à partir de ceux-ci. Ces comportements plus précis correspondent à toutes les « intersections » possibles des comportements entre eux. On définit ainsi toute une série de classes d'utilisations supplémentaires, de plus en plus précises, et qui sont des cas particuliers des comportements définis par l'utilisateur. Ces cas supplémentaires permettront, lors de l'exécution d'un appel de procédure, de retrouver le comportement qui décrit le mieux l'appel en cours, et d'obtenir ainsi la description la plus précise possible des propriétés des résultats de cet appel.

Formellement, on a :

- un ensemble de tuples abstraits « sat » de la forme $(sat_p)_p \in P$ où :
  - P est l'ensemble de tous les prédicats présents dans un programme ;
  - $\forall\ p \in P$: sat_p est un ensemble (fini) de séquences abstraites fermé par l'opération GLB.

- On peut ainsi exprimer l'ensemble des comportements Sbeh sous la forme du sat. Pour cela, on définit un **ensemble de tuples abstraits correspondant à un ensemble de comportements abstraits** Sbeh = $\langle Beh_p \rangle_{p \in P}$. Un tel ensemble de tuples abstraits est une famille sat = $\langle sat_p \rangle_{p \in P}$ de séquences abstraites telles que :

  $\forall\ p \in P$, sat_p est le plus petit ensemble :
  (i) contenant $\{B \mid \exists\ se : \langle B, se \rangle \in Sbeh_p\}$ et
  (ii) fermé par l'opération GLB, - càd - $B_1, B_2 \in sat_p \Rightarrow GLB(B_1, B_2) \in sat_p$.

La plus grande borne inférieur (glb) est une séquence abstraite qui reprend les propriétés communes de deux autres séquences abstraites B_1 et B_2. Toutes les séquences

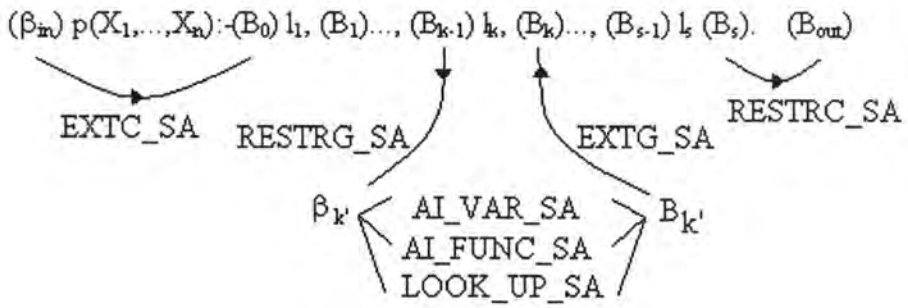
générées[5] par la concrétisation de ces deux séquences abstraites et qui sont égales doivent être générées par le glb de B_1 et B_2. C'est une opération que l'on doit redéfinir quand le domaine abstrait sera choisi. La construction du sat est réalisée par l'opération MakeSat.

7.2 FONCTIONNEMENT GÉNÉRAL DE L'ANALYSEUR

Comme nous l'avons déjà vu dans la présentation générale de l'analyseur, le fonctionnement de l'analyseur est basé sur trois types d'exécution abstraite : l'analyse du programme, l'analyse des procédures et l'analyse des clauses. Nous représentons chacune de ces analyses en allant plus en détail dans les propriétés des séquences et des substitutions abstraites utilisées. Nous commençons par l'analyse d'une clause, puis l'analyse d'une procédure et enfin l'analyse d'une clause.

7.2.1 Analyse d'une clause

Une représentation schématique de l'analyse d'une clause est :



La clause à analyser a la forme $p(X_1,...,X_n) :- l_1,...,l_s$.

La substitution β_{in} est par défaut toujours définie sur les variables de tête $X_1,...,X_n$ de la clause. C'est la *substitution abstraite en entrée* utilisée pour commencer l'exécution de la clause. Elle définit les propriétés des paramètres qui seront réellement utilisés pendant l'exécution normale du programme. Chacune des séquences abstraites B_i est définie sur toutes les variables de la clause et représente les informations capturées sur les variables de programme et les termes qui leurs sont associés suite à l'exécution des atomes $l_1,...,l_i$.

La première opération réalisée est l'extension du domaine de β_{in} à l'ensemble des variables utilisées dans la clause. L'introduction de ces nouvelles variables est représentée par la première séquence abstraite B_0.

$$B_0 = EXTC\_SA(c,\beta_{in}).$$

[5] Attention : l'utilisation du terme « générer » ne signifie pas que l'analyseur va effectivement construire toutes les séquences ou les substitutions qui répondent à un ensemble de propriétés. Cela veut seulement dire que l'on considère *n'importe quelle* séquence ou substitution concrète qui répond effectivement à ces propriétés.

A partir de cette séquence B_0, on calcule l'effet de l'exécution de chaque littéral l_k sur la séquence abstraite B_{k-1} qui résulte de l'exécution des littéraux $l_1,...l_{k-1}$ à partir de β_{in}.

Les substitutions abstraites β_k' et les séquences abstraites B_k' sont quant à elles définies sur les variables utilisées dans le littéral l_k. β_k' représente les propriétés des variables (utilisées dans l_k) résultant de l'exécution des littéraux $l_1,...,l_{k-1}$. B_k' représente l'exécution de l'atome l_k.

L'exécution du littéral l_k débute toujours par la restriction de la séquence abstraite B_{k-1} aux seules variables $X_{i_1},...,X_{i_n}$ utilisées dans l_k et par la renomination de ces variables en $X_1,...,X_n$. Ainsi l'exécution du littéral l_k se déroule toujours sur les mêmes indices 1,...,n. Cette restriction est réalisée par l'opération β_k'=RESTRG_SA(l_k,B_{k-1}).

L'exécution proprement dite de l_k dépend de sa forme :

- Si l_k a la forme $X_{i_1} = X_{i_2}$.

On a alors β_k' défini sur X_1 et X_2. On exécute l'unification de X_1 et X_2 dans β_k' par l'opération B_k'=AI_VAR_SA(β_k'). B_k' représente l'unification des deux variables à partir de β_k'.

- Si l_k a la forme $X_{i_1}=f(X_{i_2},...,X_{i_n})$.

On a alors β_k' défini sur $X_1,...,X_n$. On exécute l'unification de X_1 et $f(X_2,...,X_n)$ dans β_k' par l'opération B_k'=AI_FUNC_SA(β_k', f).

- Si l_k a la forme $q(X_{i_1},...,X_{i_m})$ où q est un symbole de prédicat.

Le littéral l_k est un appel de procédure (récursif). β_k' est définie sur les variables $X_1,...,Xm$. On recherche alors dans le sat associé à q un comportement qui définit *au moins* et *au mieux* les propriétés spécifiées dans β_k'. Cette recherche s'effectue grâce à l'opération B_k'=LOOK_UP_SA(β_k', q, sat). Si ce comportement existe, il définit les résultats de l'appel de procédure et il correspond à B_k'. Mais s'il n'existe pas, alors on a détecté un cas d'utilisation qui n'est pas repris dans les comportements de q. Dans ce cas, l'analyse de la clause va retourner un échec et les séquences abstraites B_k, $B_{k+1},...,B_s,B_{out}$ sont indéterminées.

Notons ici que dans le cas d'un appel récursif, et si le domaine abstrait qui sera choisi le permet, il est utile de vérifier la terminaison de l'appel en comparant la longueur des paramètres au moment de l'appel et leur taille au début de l'exécution de la procédure.

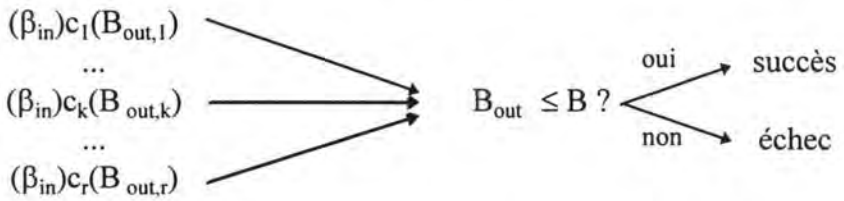
Après exécution du littéral l_k, on doit reconsidérer les variables qui ne sont pas utilisées dans l_k et vérifier si leurs propriétés n'ont pas été indirectement touchées par l'exécution de l_k. Ces opérations sont réalisées par la procédure B_k=EXTG_SA(l_k,B_{k-1},B_k'). Cette procédure, très complexe, est plus détaillée ci-dessous.

Lorsque l'exécution du littéral l_s se termine, la séquence B_s représente toutes les propriétés des termes associés aux variables de la clause. Comme on s'intéresse aux propriétés des paramètres utilisés pour exécuter la clause, on ne retient de B_s que les informations portant sur les variables $X_1,...,X_n$. C'est ce que réalise l'opération B_{out}=RESTRC_SA(c,B_s).

7.2.2 Analyse d'une procédure

L'exécution abstraite d'une procédure p se base sur l'ensemble des clauses qui constituent p, sur un comportement B associé à p et sur le sat. Cette analyse va dans un premier temps récolter les séquences abstraites $B_{out,1},...,B_{out,r}$ résultant de l'analyse de chacune des clauses $c_1,...,c_r$ qui composent p. Si une seule de ces analyses est un échec, l'analyse de la procédure est un échec. Sinon, on crée la séquence abstraite B_{out} qui représente toutes les informations contenues dans les séquences abstraites $B_{out,j}$, et on compare cette séquence abstraite à B. Si la concrétisation de B_{out} génère plus de séquences que la concrétisation de B (la procédure peut donc retourner des substitutions concrètes qui ne satisfont pas les caractéristiques définies par son comportement), alors l'analyse de la procédure est un échec. Dans le cas contraire, l'analyse est un succès.

Le déroulement de cette analyse est schématisé par :



7.2.3 Analyse d'un programme

L'analyse du programme consiste en l'analyse de toutes les procédures qui le compose en utilisant les comportements décrits par l'utilisateur pour chacune d'elles. Si toutes les analyses de procédure sont des succès, alors l'analyse du programme est un succès. Sinon, c'est un échec.

7.3 DESCRIPTION DES OPÉRATIONS PRINCIPALES

7.3.1 Extension d'une séquence abstraite aux variables d'une clause

EXTC_SA(c,β) = B'.

Cette opération est la première réalisée lorsqu'on interprète une clause c à partir d'une substitution abstraite β. Elle a pour but de construire la première séquence abstraite B_0 qui servira de base à l'interprétation de tous les atomes de la clause c. Elle réalise deux choses : elle étend le domaine de β aux variables de programme présentes dans c mais qui ne sont pas des variables de tête, et elle construit une représentation sous forme de séquence abstraite de cette extension.

On suppose que c est syntaxiquement représenté par $pr(X_1,...,X_n)$:- .... Toute substitution abstraite β est définie sur l'ensemble des variables de tête de c, c'est à dire $X_1,...,X_n$.

Mais il peut aussi y avoir d'autres variables de programme présentes dans c et qui ne sont pas des variables de tête. On désigne par $X_{n+1},...,X_m$ cet ensemble de variables ($m \geq n$). Or, par convention, toute séquence abstraite B_i représentée dans le fonctionnement de l'interprétation d'une clause est définie sur toutes les variables de c. On doit donc compléter β en y insérant les variables $X_{n+1},...X_m$ et les informations qui leurs sont propres. Ensuite, on doit construire une représentation sous forme de séquence abstraite (B') de l'introduction de ces nouvelles variables.

Cette opération respecte les propriétés définies par :

Si (1) $dom(\theta) = D' = \{X_1,...,X_n, X_{n+1},...,X_m\}$
(2) $\theta_{|\{X_1,...,X_n\}} \in Cc(\beta)$
(3) $X_{n+1}\theta$,...,$X_m\theta$ sont des variables de renomination distinctes
(4) $X_{n+j}\theta \notin codom(\theta_{|\{X_1,...,X_n\}})$ $(1 \leq j \leq m-n)$
Alors $\langle \theta_{|\{X_1,...,X_n\}}, \langle \theta \rangle \rangle \in Cc(B')$.

Ces propriétés nous disent que l'ensemble des séquences concrètes générées par B' ont toutes une substitution concrète en entrée définie sur les variables de tête de c et qui respecte les propriétés spécifiées par β[6]. De plus, la séquence de substitutions est composée d'une seule substitution concrète θ définie sur toutes les variables de la clause c. Cette substitution θ est identique à la substitution en entrée de la séquence quand on ne considère que les variables de tête. De plus, les termes associés aux variables $X_{n+1},...,X_m$ sont des variables distinctes qui n'apparaissent pas dans les termes associés aux variables $X_1,...,X_n$.

Par exemple, on a θ={X_1/a, X_2/f(b),X_3/Y} issue de la concrétisation de β. On veut y insérer les variables X_4 et X_5. On obtient la séquence :

B'=⟨ θ, ⟨{X_1/a, X_2/f(b), X_3/Y, X_4/VAR4, X_5/VAR5}⟩⟩.

La séquence B'', elle, n'est pas correcte à cause du terme associé à X_1 et de la variable associée à X_5. B''=⟨ θ, ⟨{X_1/A, X_2/f(b), X_3/Y, X_4/VAR4, X_5/Y}⟩⟩.

7.3.2 Restriction d'une séquence abstraite aux termes d'un atome

RESTRG_SA(l_k, B_{k-1}) = β'.

Cette opération est utilisée quand on est sur le point d'exécuter le littéral l_k à partir du résultat de l'exécution des littéraux $l_1,...,l_{k-1}$. Ce résultat est représenté par la séquence abstraite B_{k-1}.

Cette séquence abstraite B_{k-1} est définie sur toutes les variables $X_1,...,X_n$ de la clause. Et donc, chacune des substitutions concrètes appartenant aux séquences générées par B_{k-1} est définie sur toutes ces variables. Or, nous avons vu que l'exécution de l'atome l_k se fait toujours sur les seules variables présentes dans l_k. De plus, elle ne se fait que sur des variables indicées de manière continue.

Si on réfléchit au niveau concret, il faut donc (i) projeter les variables (utilisées dans l_k) de toutes les substitutions concrètes appartenant aux séquences générées par la

[6] Cette substitution en entrée est générée par la concrétisation de β.

concrétisation de B_{k-1} et (ii) renommer ces variables de manière continue dans chacune des substitutions retournées par la projection. On obtient ainsi l'ensemble des substitutions concrètes qui sont générées par β'.

En supposant que $\{X_{i_1},...,X_{i_j}\}$ est l'ensemble des variables utilisées dans l_k, l'opération respecte la propriété suivante :

$$\forall \langle\theta,S\rangle \in Cc(B_{k-1}),\ \forall\ \theta' \in Subst(S) : \theta'=\{X_1/t_1,...,X_n/t_n\} \Rightarrow \{X_1/t_{i_1},...,X_j/t_{i_j}\} \in Cc(\beta')$$

où l'on a que, à partir de toutes les séquences ⟨θ,S⟩ générées par la concrétisation de B_{k-1}, la projection des variables $X_{i_1},...,X_{i_j}$ de toutes les substitutions concrètes appartenant à S retourne un ensemble E de substitutions concrètes, toutes définies sur $X_{i_1},...,X_{i_j}$. Si pour chacune des substitutions de E, on renomme[7] les variables $X_{i_1},...,X_{i_j}$ respectivement en $X_1,...,X_j$, on a l'ensemble des substitutions concrètes générées par β'.

Exemple :
si l_k a la forme $X_5=X_2$ et que $\theta'=\{X_1/a, X_2/f(h(g(T))), X_3/Y, X_4/h(g(T)), X_5/Z\}$ est une des substitutions générées par la concrétisation de B_{k-1}, on a $E=\{..., \{X_5/Z, X_2/f(h(g(T)))\}, ...\}$ et, après renomination, on a les substitutions $\{..., \{X_1/Z, X_2/f(h(g(T)))\}, ...\}$ qui ne contiennent que l'information utile pour réaliser l'unification l_k et qui sont toutes générées par β'.

7.3.3 Interprétation d'un atome

7.3.3.1 Unification de variables dans une séquence abstraite

AI_VAR_SA(β) = B.

Cette opération réalise l'unification de deux variables $X_1=X_2$. Remarquons ici que ces variables sont toujours identifiées par ces mêmes indices 1 et 2, et que le domaine de la substitution abstraite β dans laquelle va se dérouler cette unification est toujours $\{X_1, X_2\}$.

La substitution abstraite β représente l'ensemble des substitutions concrètes dans lesquelles on peut réaliser cette unification, alors que la séquence abstraite B représente l'ensemble des séquences concrètes qui résultent de cette unification. Comme l'unification de deux variables ne réussit qu'au maximum une fois, la suite des substitutions concrètes représentée par cette séquence a toujours une longueur maximale de 1.

Cette opération respecte la condition suivante :

$$\theta \in Cc(\beta) \wedge \sigma \in mgu(X_1\theta\ ,\ X_2\theta) \Rightarrow \langle\theta,\langle\theta\sigma\rangle\rangle \in Cc(B)$$
$$\theta \in Cc(\beta) \wedge mgu(X_1\theta\ ,\ X_2\theta) = \varnothing \Rightarrow \langle\theta,\langle\rangle\rangle \in Cc(B)$$

L'unification peut soit réussir soit échouer. La réussite de l'unification dépend des termes associés aux deux variables.

---

[7] On renomme seulement dans la partie gauche des associations X_i/t_i.

Réfléchissons toujours au niveau concret et sur des substitutions θ générées par la concrétisation de β. Dans le cas où l'unification réussit, on connaît alors un des mgus[8] σ de l'unification des termes associés aux variables X_1 et X_2 dans une substitution θ spécifique. Si on compose θ avec ce mgu σ, on obtient une nouvelle substitution concrète qui associe à X_1 et X_2 le terme résultant de l'unification. On représente le passage de θ à cette nouvelle substitution pas la séquence $\langle\theta,\langle\theta\sigma\rangle\rangle$. Si l'unification réussit pour une substitution θ donnée générée par β, elle réussit pour n'importe quelle θ généré par β. Cette séquence est générée par B.

Si l'unification échoue (il n'y a pas de mgu), cela signifie d'un point de vue concret (i) qu'il n'y a pas de substitution résultant de l'unification et (ii) que, si cette unification fait partie d'une clause et qu'elle est suivie d'autres atomes, alors il est impossible de poursuivre l'exécution de cette clause. Dans le cadre de l'analyse de programmes, cette impossibilité *est* une information : on sait que pour n'importe laquelle des substitutions concrètes générées par β, l'unification des variables X_1 et X_2 échouera. On reporte cette information par la séquence $\langle\theta,\langle\rangle\rangle$ qui nous indique qu'aucune solution n'existe suite à cette unification.

Par exemple, l'unification de X_1 et X_2 à partir de la substitution $\theta=\{X_1/g(b), X_2/f(g(b))\}$ générée par β va retourner la séquence concrète $\langle\theta,\langle\rangle\rangle$ qui devra être générée par B. Par contre la même unification dans la substitution $\theta'=\{X_1/f(g(b)), X_2/f(W)\}$ va réussir et retourner la séquence $\langle\theta,\langle\{X_1/f(g(b)),X_2/f(g(b))\}\rangle\rangle$ où X_1 et X_2 partagent la même valeur, et qui devra être générée par B.

7.3.3.2 Unification d'une variable et d'un foncteur dans une séquence abstraite.

AI_FUNC_SA(β,f) = B.

Cette opération est très similaire à l'opération AI_VAR_SA. Elle réalise l'unification d'une variable X_1 à un terme $f(X_2,..X_n)$. Remarquons ici que ces variables sont toujours identifiées par les mêmes indices 1,...,n, et que le domaine de la substitution abstraite β dans laquelle va se dérouler l'unification est toujours $\{X_1,...,X_n\}$.

Comme pour l'opération AI_VAR_SA, la substitution abstraite β représente l'ensemble des substitutions concrètes dans lesquelles on peut réaliser l'unification, alors que la séquence abstraite B représente l'ensemble des séquences concrètes qui résultent de cette unification. Iic, aussi, la suite des substitutions concrètes représentées par ces séquences a toujours une longueur maximale de 1.

L'opération respecte la condition suivante :

$$\theta \in Cc(\beta) \wedge \sigma \in mgu(X_1\theta, f(X_2,...,X_n)\theta) \Rightarrow \langle\theta,\langle\theta\sigma\rangle\rangle \in Cc(B)$$
$$\theta \in Cc(\beta) \wedge mgu(X_1\theta, f(X_2,...,X_n)\theta) = \varnothing \Rightarrow \langle\theta,\langle\rangle\rangle \in Cc(B)$$

[8] Rappelons qu'il peut y avoir plusieurs mgu satisfaisant une unification.

L'unification peut soit réussir soit échouer. La réussite de l'unification dépend des termes associés aux variables.

Réfléchissons à partir des substitutions θ générées par la concrétisation de β. Dans le cas où l'unification réussit[9], on connaît alors un des mgus σ de l'unification des termes associés aux variables $X_1,...,X_n$ dans la substitution θ spécifique. Si on compose θ avec ce mgu σ, on obtient une nouvelle substitution concrète qui associe à X_1 un terme de la forme f/n-1 et dont les termes composés sont le résultat de l'unification.

Comme pour l'opération AI_VAR_SA, on représente le passage de θ à cette nouvelle substitution pas la séquence ⟨θ,⟨θσ⟩⟩. Cette séquence est générée par B.

Si l'unification échoue, le raisonnement est identique à celui détaillé pour l'opération AI_VAR_SA.

Par exemple, l'unification de X_1 et $f(X_2,X_3)$ à partir de la substitution $\theta=\{X_1/g(b), X_2/f(g(b)), X_3/h\}$ générée par β va retourner la séquence concrète ⟨θ,⟨⟩⟩ qui devra être générée par B. Par contre, la même unification dans la substitution $\theta'=\{X_1/f(f(b),X), X_2/f(W), X_3/W\}$ va réussir et retourner la séquence ⟨θ,⟨{$X_1/f(f(b),X)$, $X_2/f(b)$, X_3/b}⟩⟩ qui devra être générée par B.

7.3.3.3 Exécution d'un appel de procédure.

LOOKUP_SA(β, p, sat) = ⟨success, B⟩.

Lorsque le littéral l_k à exécuter a la forme $p(X_{i_1},...,X_{i_n})$ et que la substitution abstraite en entrée pour l'exécution de ce littéral est β, on a le raisonnement suivant.

β représente l'ensemble de toutes les substitutions concrètes dans lesquelles peut avoir lieu l'appel à la procédure p. Les paramètres utilisés sont les termes associés aux variables $X_{i_1},...,X_{i_n}$ dans ces substitutions concrètes. Ces paramètres ont certaines propriétés (définies par β), et c'est sur base de celles-ci que l'on va résoudre l'exécution au niveau abstrait. En effet, le sat associé à p contient la description de tous les appels possibles de la procédure p pour lesquels un comportement est défini. Il devrait donc contenir au moins une description des propriétés des paramètres utilisés pour exécuter p. Si ces descriptions existent, elles définissent aussi les propriétés des paramètres après l'exécution de p.

On va donc rechercher dans le sat la description des propriétés des paramètres de p la plus proche possible de l'appel l_k. Cette description doit *au moins* définir les propriétés des paramètres utilisés. Mieux elle définit ces propriétés, meilleurs seront les résultats de l'exécution de l_k. Si elle existe (on la note B), on considère que les résultats l_k ont les propriétés définies par la description trouvée, et l'exécution de l_k est représentée par la séquence B.

On le voit, on n'exécute pas l'interprétation de p pour connaître le résultat de l'exécution de l_k. On va seulement rechercher dans l'ensemble des comportements de p celui

[9] Le terme associé à X_i dans θ est soit une variable soit un terme de la forme f/n-1.

qui correspond le mieux à l'appel actuel. Si cet appel n'est pas repris dans les comportements, on a détecté une utilisation de procédure qui n'est pas vérifiée par les comportements, et l'analyse est un échec.

L'opération respecte la propriété suivante :

$\exists B' \in sat_p : \beta \leq input(B') \Rightarrow success = true \wedge B = min \{B' \mid B' \in sat_p, \beta \leq input(B')\}$.

Le sat associé à la procédure p peut contenir plusieurs comportements qui définissent au moins les propriétés de l'appel courant définies dans β. Ces comportements constituent B'. On retient de ceux-ci celui (B) qui définit le mieux β (la concrétisation de ce comportement génère un ensemble de séquences concrètes dont les substitutions en entrée correspondent à une des substitutions générées par la concrétisation de β). Cet ensemble de séquences concrètes est égal ou inclus dans la concrétisation de tout autre comportement B'. Si un tel B existe, il constitue le résultat de l'exécution de l_k et la valeur *success* représente le fait que l_k réussit à partir de β. Si B n'existe pas, l'analyse est un échec.

Notons ici que dans le cas d'un appel récursif, il *peut* être utile de vérifier si cet appel de procédure se termine. Cependant, cette vérification ne peut se faire qu'une fois le domaine abstrait choisi. Or, cet algorithme ignore la composition des séquences et des substitutions abstraites. En ce moment, on ne peut donc pas dire si on vérifie ou non la terminaison de l'appel, et si oui, on ignore comment.

7.3.4 Extension d'une séquence abstraite aux variables d'une clause après unification.

$EXTG\_SA(l_k, B_{k-1}, B_{aux}) = B_k$.

Cette opération est utilisée après l'exécution du littéral l_k. Elle calcule l'effet de cette exécution (qui est donné par la séquence abstraite B_{aux}) sur la séquence abstraite B_{k-1}.

On suppose que l_k utilise les variables $X_{i_1},...,X_{i_m}$. B_{k-1} est la séquence abstraite résultant de l'exécution de $l_1,...,l_{k-1}$. Elle est définie sur toutes les variables de la clause, soit $X_1,...,X_n$.

La substitution abstraite β' utilisée pour résoudre l_k est toujours le résultat de $RESTRG\_SA(l_k,B_{k-1})$ et est définie sur les variables $X_1,...,X_m$. B_{aux} est le résultat de l_k à partir de β'. B_{aux} est donc elle aussi définie sur les variables $X_1,...X_m$.[10]

Il s'agit maintenant de calculer la séquence abstraite B_k. Cette séquence doit être définie sur toutes les variables de la clause. Elle doit d'une part contenir les nouvelles propriétés des variables $X_{i_1},...,X_{i_m}$ issues de l'exécution de l_k, et d'autre part contenir les

---

[10] Cependant, la substitution en entrée de B_{aux} peut être différente de celle calculée par RESTRG_SA. En effet, si l_k a la forme $X_{i_1}=X_{i_2}$ ou la forme $X_{i_1}=f(X_{i_2},...,X_{i_m})$, on a toujours que $\beta=RESTRG\_SA(l_k,B_{k-1})$. Si l_k a la forme $p(X_{i_1},...,X_{i_m})$, β est la substitution en entrée de la séquence B_{aux} trouvée par la procédure LOOK_UP_SA. Comme la substitution en entrée de B_{aux} décrit au moins les propriétés définies dans β, on a $\beta > RESTRG\_SA(l_k,B_{k-1})$.

propriétés définies dans B_{k-1} pour les variables qui ne sont pas utilisées dans l_k. *Mais il faut aussi tenir compte des liens qui existaient entre toutes les variables dans la séquence* B_{k-1}. La raison est illustrée ci-dessous.

Réfléchissons tout d'abord au niveau concret. Si les variables X_1 et X_2 sont unifiées lors de l'exécution d'un des littéraux $l_1,...,l_{k-1}$[11], alors elles seront toujours associées au même terme en même temps. Donc, tout changement apporté sur le terme associé à l'une de ces variables vaut également pour l'autre variable. Supposons que X_1 et X_2 sont associées à la même variable. Si seule X_1 est utilisée dans l_k, et que l'exécution de l_k modifie le terme associé à X_1 (par exemple, X_1 n'est plus associé à une variable mais à un terme clos) et que l'on ne répercute pas ce changement sur X_2, on aura *au niveau abstrait* que X_1 ne peut être associé qu'à des termes clos et que X_2 ne peut être associé qu'à des variables. Ce qui signifie que plus aucune terme ne peut être associé aux deux variables en même temps, *alors que ces deux variables partagent toujours le même terme.*

Si on en revient à l'opération EXTG_SA, on a que B_{aux} n'est définie que sur les variables utilisées par l_k (X_1 dans l'exemple). On ne peut donc pas se contenter de calculer B_k en reprenant les informations de B_{k-1} pour les variables qui ne sont pas utilisées dans l_k et les informations de B_{aux} pour les variables utilisées dans l_k. Il faut en plus propager les effets de l'unification sur toutes les variables de la clause qui ne sont pas utilisées par l_k mais qui sont liées d'une manière quelconque à une des variables utilisées dans l_k.

L'opération doit respecter les propriétés suivantes :

Si $\langle\theta,S\rangle \in Cc(B_{k-1})$
$S = \langle\theta_1,...,\theta_q\rangle$
$\theta_j = \{X_1/t_{1_j},...X_n/t_{n_j}\}, j \in \{1,...,q\}$
$\theta_j' = \{X_1/t_{i_{1_j}},...X_n/t_{i_{m_j}}\}, j \in \{1,...,q\}$
$\langle\theta_j', S_j'\rangle \in Cc(B_{aux}), j \in \{1,...,q\}$
$S_j = AUX(l_k, \theta_j, S_j'), j \in \{1,...,q\}$
Alors $\langle\theta, S_1::...::S_q\rangle \in Cc(B')$
où $AUX(l_k, \theta, S) = S'$ est définie par :
$\theta = \{X_1/t_1,...,X_n/t_n\}$
$\theta' = \{X_1/ t_{i_1},...,X_m/t_{i_m}\}$
$S=\langle\theta_1,...,\theta_q\rangle$
$\exists \sigma_j : \theta_j = \theta'\sigma_j$, $dom(\sigma_j) \subseteq codom(\theta'), j \in \{1,...,q\}$
$\Rightarrow$ $S'=\langle\theta\sigma_1,...,\theta\sigma_q\rangle$

B_{k-1} génère toutes les séquences concrètes calculées par les littéraux $l_1,...,l_{k-1}$. Chacune des substitutions θ_j de ces séquences peut être utilisée pour exécuter l_k. Elle est définie sur toutes les variables de la clause. Pour exécuter l_k, on projette les variables de θ_j qui sont utilisées par l_k et on obtient une substitution θ_j' dans laquelle on exécute l_k. Le résultat de cette exécution est une séquence de substitutions concrètes S_j'. Si on considère toutes les substitutions θ_j et que l'on répète ce raisonnement, on a toutes les substitutions qui résultent de l'exécution de l_k à partir de toutes les séquences résultant de l'exécution de $l_1,...,l_{k-1}$. Donc

[11] Cette information est alors présente dans B_{k-1}.

B_{aux} définit toutes les substitutions retournées par l'exécution des littéraux $l_1,...,l_k$ *et définies sur les variables utilisées dans* l_k.

Il faut alors reprendre chacune de ces séquences résultats et y réintégrer l'information sur les variables de la clause non utilisées dans l_k tout en propageant les effets de l'exécution de l_k. C'est ce qui est réalisé par l'opération AUX.

AUX construit une suite de substitutions concrètes toutes définies sur toutes les variables de la clause $X_1,...,X_n$. Elle construit cette liste sur base d'une suite de substitutions S_j' retournées par l'exécution de l_k et qui sont toutes définies[12] sur les variables $X_{i_1},...,X_{i_m}$. Pour chacune de ces substitutions, on recherche une substitution de renomination σ_j qui ne contient que les variables présentes dans les termes associés aux variables $X_{i_1},...,X_{i_m}$ et différentes de celles-ci, et on compose cette substitution σ_j avec chacune des substitutions de S_j'. Cette composition a pour effet d'intégrer les résultats de l_k dans une substitution[13] définie sur $X_1,...,X_n$ et d'y propager les effets de l'exécution de l_k. Si on répète ces opérations pour toutes les substitutions en entrée utilisées pour exécuter l_k, on obtient toutes les séquences résultant de l'exécution des littéraux $l_1,...,l_k$ et représentées par B'.

Par exemple, le littéral l_k *a la forme* $X_1=X_3$. *La concrétisation de* B_{k-1} *génère entre autres la séquence* $\langle\theta_{in}, \langle\{X_1/a,X_2/g(W),X_3/W,X_4/W\}\rangle\rangle$ *où* $\theta_{in}=\{X_1/A,X_2/B,X_3/C,X_4/D\}$. B_{aux} *vaut après l'unification* $\langle\{X_1/a,X_3/W\}, \langle\{X_1/a, X_3/a\}\rangle\rangle$. *Si on se contente de reporter ces valeurs dans la séquence* B_k, *on a* $B_k=\langle\theta_{in}, \langle\{X_1/a,X_2/g(W),X_3/a,X_4/W\}\rangle\rangle$ *qui n'est pas correcte : on sait que* X_2 *et* X_4 *sont indirectement concernés par cette unification. Or dans* B_k, *le résultat ne pose aucun problème pour* X_3, *mais en pose un pour* X_2 *et* X_4. *Ces variables sont toujours associées au terme W alors qu'elles partageaient le même terme que* X_3 *qui lui, maintenant est associé au terme a. La séquence* B_k *correcte est* $\langle\theta_{in}, \langle\{X_1/a,X_2/g(a),X_3/a,X_4/a\}\rangle\rangle$.

7.3.5 Restriction d'une séquence abstraite aux variables d'une clause après unification

RESTRC_SA(c,B) = B'.

Cette opération est la dernière réalisée lorsqu'on exécute une clause. Elle a pour but de réduire la séquence abstraite résultant de l'exécution de la clause c aux variables de la tête de c.

B est la séquence abstraite qui caractérise tous les résultats retournés par l'exécution de c. Elle est définie sur toutes les variables c. Or, on ne s'intéresse qu'aux propriétés des variables de tête de c. Il faut donc réduire cette séquence abstraite aux seules variables de tête.

---

[12] Après renomination.

[13] Qui résulte de l'exécution de $l_1,...,l_{k-1}$.

Cette opération de restriction doit respecter la condition suivante :

$\langle\theta,S\rangle \in Cc(B) \Rightarrow \langle\theta, S_{|dom_{in(B)}}\rangle \in Cc(B')$.

Toutes les séquences $\langle\theta,S\rangle$ générées par la concrétisation de B sont définies sur les variables de la clause. La substitution concrète en entrée de chacune de ces séquences est quant à elle définie sur les seules variables de tête de c. Par contre toutes les substitutions concrètes d'une séquence particulière sont définies sur toutes les variables de c. On projette alors chacune de ces substitutions sur les variables de tête de c. Si on effectue cette projection sur toutes les substitutions de toutes les séquences générées par B, on a toutes les séquences générées par B'.

Par exemple, une séquence générée par la concrétisation de B est $\langle\{X_1/A,X_2/B,X_3/C,X4/D\},\langle\{X_1/f(a,g(b)),X_2/g(b),X_3/b,X_4/f(c)\}\rangle$. *Le résultat de la restriction aux variables de tête* X_1 *et* X_2 *retourne la séquence* $\langle\{X_1/A,X_2/B\},\langle\{X_1/f(a,g(b)),X_2/g(b)\}\rangle$ *qui doit être générée par B'.*

7.3.6 Analyse d'une clause

AnalyseClause(β_{in}, c, se) = $\langle$success, $B_{out}\rangle$.

Cette opération vérifie que la clause c de la procédure p respecte les contraintes définies dans les comportements qui est associé à p (quand elle est exécutée à partir de β_{in}). Se est utilisée pour calculer les tailles des termes .

β_{in} est une substitution abstraite en entrée définie sur les variables de tête de la clause c. L'analyse de la clause va prendre en charge l'extension de β_{in} à toutes les variables de c, et va exécuter successivement les littéraux qui compose c sur base de cette substitution abstraite étendue. Cette exécution se poursuit tant que l'exécution de chaque littéral est un succès. Le résultat de cette analyse est la description des propriétés de toutes les substitutions concrètes qui seront retournées par la clause si son exécution réelle se termine.

L'opération respecte la condition :

Success = true $\Leftrightarrow$
1) si t = $\langle\theta,B,se,p\rangle$,
c est la clause $p(X_1,...,X_n)$:-$l_1,...,l_s$,
$K \in \{0,...,s-1\}$,
l_{k+1} est l'appel (récursif) à une procédure $q(X_{i_1},...,X_{i_m})$,
$\langle \theta,\langle l_1,...,l_k \rangle,c \rangle \rightarrow_t S$, S est la séquence de
substitutions résultats de l'exécution de $l_1,...,l_k$ avec θ,
$\theta' \in Subst(S)$,
$\theta'' = \{X_1/(X_{i_1}\theta'),...,X_m/(X_{i_m}\theta')\}$
alors
$\exists (B',se') \in Beh_q : \langle\theta',B',se',q\rangle < \langle\theta,B,se,p\rangle$
2) $\langle\theta,c\rangle \rightarrow_t S \Rightarrow \langle\theta,S\rangle \in Cc(B_{out})$.

La première partie nous dit que chaque appel de procédure atteint pendant l'exécution de la clause c à partir de θ utilise des paramètres plus précis que les termes utilisés

au début de l'exécution de c. Tous les sous appels sont soit des appels à d'autres procédures, soit des appels récursifs avec une taille décroissante pour les paramètres. De plus, la séquence obtenue après l'exécution de chaque littéral est plus précise que la séquence abstraite initiale. Si l'exécution du littéral l_k réussit, alors la séquence abstraite B_k est définie par un comportement au moins.

La deuxième partie dit que toute l'information sur l'exécution de la clause c avec θ qui peut être déduite de ce que Sbeh affirme sur les appels de procédure plus précis (-càd- toute l'information donnée par l'extension $\rightarrow_t$) est contenue dans B_{out}. Toutes les séquences retournées par l'exécution ont leurs propriétés définies par B_{out}.

De plus, si S est une séquence finie, on peut prouver que l'exécution de chaque appel récursif $l_i=p(X_{i_1},...,X_{i_n})$ se termine en utilisant se :

Si $\theta \in Cc(\beta_{in})$,
$\langle \theta, l_1,...,l_{i-1} \rangle \rightarrow S'$, S' est le résultat de l'exécution
des littéraux $l_1,...,l_{i-1}$ avec θ,
$l_i=p(X_{i_1},...,X_{i_n})$ un appel récursif à la procédure en exécution p,
$\theta' \in S'$
Alors $[[se]] \langle ||X_1\theta||,...,||X_n\theta|| \rangle > [[se]] \langle ||X_{i_1}\theta'||,...,||X_{i_n}\theta'|| \rangle$.

Cette condition dit que la taille des variables au début de l'exécution de la clause est supérieure à la taille des variables correspondantes utilisées lors de l'exécution de l_i.

7.3.7 Analyse d'une procédure

AnalyseProcedure (B,p,se) = success.

L'analyse d'une procédure p a pour but de vérifier si l'exécution de cette procédure satisfait les propriétés définies dans le comportement B et les relations entre la taille des termes et le nombre de solutions définies dans se.

L'opération vérifie la propriété :

Si success est vrai alors on a, $\forall \theta \in Cc(input(B))$
1) $\forall c \equiv p(X_1,...,X_n) :- l_1,...,l_s$,
si $t = \langle \theta,B,se,p \rangle$ représente les conditions d'exécution
de la procédure p,
$k \in \{0,...,s-1\}$,
$l_{k+1} \equiv q(X_{i_1},...,X_{i_m})$,
$(\theta,\langle l_1,...,l_k \rangle,c) \rightarrow_t S$, S est la séquence de
substitutions résultat de l'exécution de $l_1,...,l_k$ avec θ,
$\theta' \in Subst(S)$,
$\theta''=\{X_1/(X_{i_1}\theta'),...,X_m/(X_{i_m}\theta')\}$ le résultat de $c\theta$,
alors $\exists \langle B',se' \rangle \in Beh_q : \langle \theta'',B',se',q \rangle < \langle \theta,B,se,p \rangle$
2) $\langle \theta,pr \rangle \rightarrow_t S \Rightarrow \langle \theta,S \rangle \in Cc(B)$ où pr est le texte définissant p.

Toute exécution réussie de la procédure p à partir d'une substitution concrète générée par la substitution abstraite en entrée définie dans le comportement B répond à deux conditions. La première condition nous dit que pour n'importe quelle clause, la séquence obtenue après l'exécution de chaque littéral est plus précise que la séquence abstraite initiale. Si l'exécution du littéral l_k réussit, alors la séquence abstraite B_k est définie par un comportement au moins. La seconde condition nous dit que toutes les séquences obtenues pendant l'exécution sont des instances de la séquence initiale. De plus, se est utilisé pour prouver la terminaison lors des appels récursifs pendant l'exécution.

7.3.8 Analyse d'un programme

AnalyseProgram(P,Sbeh) = success.

Cette opération a pour but d'exécuter un programme P et de vérifier si chacune des procédures qu'il contient répond aux propriétés des comportements données par l'utilisateur (Sbeh).

L'analyse du programme est un succès *(success)* si chaque procédure de P a le comportement attendu pour cette classe quand elle est exécutée pour une classe de substitutions décrite dans un comportement particulier de cette procédure. Dans ce cas, Sbeh décrit l'exécution de tous les appels de procédures autorisés par lui, et tous ces appels se terminent.

L'opération respecte la condition suivante :

$$\text{success} = \text{vrai} \Leftrightarrow \forall \langle p, \{\langle B_1,se_1\rangle,...,\langle B_q,se_q\rangle\}\rangle \in \text{Sbeh} : \forall i : i \in \{1,...q\} :$$
$$\theta \in \text{Cc}(\text{input}(B_i)) \wedge \langle\theta,p\rangle \rightarrow S \Rightarrow \langle\theta,S\rangle \in \text{Cc}(B_i).$$

Le succès d'une procédure p est vérifié si, pour chacun de ses comportements défini dans Sbeh, son exécution à partir d'une substitution concrète en entrée générée par ce comportement retourne une séquence de substitutions qui possède les propriétés définies par ce comportement. Si toutes les procédures du programme sont vérifiées, alors l'analyse du programme est un succès. Sinon, elle est un échec.

7.3.9 Construction du sat - MakeSat

MakeSat(Sbeh) = sat.

Cette opération retourne un ensemble de tuples abstraits sat = $(sat_p)_{p \in P}$ représentant l'information contenue dans l'ensemble des comportements Sbeh. On suppose que le programme est composé d'un ensemble de procédures p pour lesquelles l'utilisateur a défini au moins un comportement.

La construction du sat doit respecter la propriété suivante :

$$\text{Si} \quad \langle p, \{\langle B_1,se_1\rangle,...,\langle B_q,se_q\rangle\}\rangle \in \text{Sbeh}$$
$$i_1,...,i_n \in \{1,...,q\}$$

$$B = GLB(...(GLB(B_{i_1}, B_{i_2}),...), B_{i_n})$$

Alors $B \in sat_p$.

Toutes les fermetures greatest lower bound (glb) possibles des séquences abstraites définies dans les comportements d'une procédure retournent une séquence abstraite qui est elle-même présente dans le sat de cette procédure.

8. Le domaine abstrait

Ce chapitre définit de manière formelle le domaine abstrait utilisé dans l'analyseur. On y retrouvera la spécification des structures utilisées par l'algorithme générique (les séquences abstraites et les substitutions abstraites) ainsi que les différents types de propriété que l'on veut capturer sur les programmes Prolog.

Nous commencerons par décrire la structure des séquences abstraites parce que c'est elle qui représente le *déroulement* de l'exécution d'une clause et d'une procédure[1]. C'est aussi à l'aide de cette structure que les comportements des procédures peuvent être représentés et utilisés. Nous verrons également la représentation choisie pour calculer les relations entre les tailles des termes entre eux dans une même substitution abstraite, les tailles des termes correspondants dans différentes substitutions abstraite et celles définie entre les tailles des termes et le nombre de solutions d'une procédure.

Nous définirons ensuite la notion de substitutions abstraites qui a été retenue. Cette structure est semblable à celle utilisée dans le système GAIA : il est donc possible de représenter les modes, les formes, les partages de variables et de valeurs pour tous les termes présents dans la substitution. Mais de nouvelles informations ont été ajoutées et il est à présent possible de représenter les types des termes et les relations

[1] Et que ceci constitue la nouveauté de cet analyseur

entre les tailles des termes. Nous verrons que la représentation des relations entre les tailles des termes peut se baser sur la même structure que celle utilisée dans les séquences abstraites. Nous verrons aussi que le calcul du types d'un terme tient compte du mode et de la forme de celui-ci.

8.1 SÉQUENCE ABSTRAITE

Rappelons[2] que la séquence abstraite permet de représenter le *déroulement* de l'exécution d'une clause ou d'une procédure. Elle renseigne sur les modifications apportées sur les propriétés des paramètres utilisés lors de cette exécution et sur *l'ensemble* des solutions qui peuvent être retournées celle-ci.

On définit d'abord une pseudo séquence abstraite. On définit ensuite un ensemble de contraintes qui, si elles sont vérifiées par une pseudo séquence abstraite, font que celle-ci est une séquence abstraite. La pseudo séquence abstraite est utilisée dans l'implantation parce qu'elle facilite certaines opérations en autorisant une représentation non « *optimale* » d'une séquence de substitutions. Une opération CleanUp est utilisée pour appliquer les contraintes qui rendent une pseudo séquence abstraite réellement abstrait.

Une (pseudo) séquence abstraite va utiliser les substitutions abstraites et les relations sur les tailles. Comme ces deux notions n'ont pas encore été définies, on va spécifier les séquences abstraites sur base des ensembles de séquences concrètes générées par leur concrétisation[3].

Une pseudo séquence abstraite est soit $\perp$ soi un 5-tuple de la forme $\langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$ où :

- β_{in} est une substitution abstraite définie sur les variables $X_1,...,X_n$;
- β_{ref} est une substitution abstraite définie sur les variables $X_1,...,X_n$;
- β_{out} est une substitution abstraite définie sur l'ensemble des variables $X_1,...,X_m \subseteq \{X_1,...,X_n\}$;
- E_{ref_out} est une relation sur les tailles définies sur tous les termes et sous-termes de β_{ref} et β_{out} ;
- E_{sol} est une relation sur les tailles définies sur tous les termes de β_{out} et sur une variable *sol*.

Par convention, une séquence abstraite est notée B et une substitution abstraite est notée β.

---

[2] Une introduction plus complète a déjà été donnée dans la présentation générale de l'analyseur.
[3] Tout comme on a défini l'algorithme abstrait sur seule base des fonctions de concrétisation.

On notera Input(B) = β_{in}, Output(B) = β_{out}, $dom_{in}(B) = dom(\beta_{in})$ le domaine de définition de β_{in} , $dom_{out}(B) = dom(\beta_{out})$ le domaine de définition de β_{out}.

β_{in} est la substitution abstraite en entrée qui définit les propriétés des termes concrets qui peuvent être utilisés comme paramètres pour exécuter un atome, une clause ou une procédure. Elle définit donc une classe d'appels particuliers. Elle est définie sur l'ensemble des variables de programme $X_1,...,X_n$.

β_{ref} est une substitution abstraite plus raffinée que β_{in}. Elle définit les propriétés des paramètres tels que l'exécution d'un atome, d'une clause ou d'une procédure se termine au moins une fois. Elle caractérise donc un sous-ensemble des termes concrets générés par la $Cc(\beta_{in})$[4]. Pour cette raison, elle est définie sur le même ensemble de variables que β_{in}. Nous dirons qu'elle définit les propriétés des paramètres pour lesquels on est sûr que l'atome, la clause ou la procédure se termine au moins une fois.

Remarque : il ne s'agit pas de l'approximation exacte de toutes les substitutions concrètes[5] qui mènent au succès d'une exécution. Il se peut en effet que certaines substitutions concrètes qui répondent aux propriétés définies dans β_{ref} ne mènent pas au succès de l'exécution, mais si β_{ref} est différente de β_{in}, *il y en aura moins* par rapport à celles qui satisfaisont β_{in} et pour lesquelles l'exécution échoue. Cependant, par abus de langage, nous continuerons à dire qu'elle définit les propriétés des paramètres qui mènent *certainement* à un succès au moins une fois.

Par convention, nous appelerons la substution abstraite β_{ref} plus raffinée que β_{in} **la substitution raffinée.**

β_{out} est une substitution abstraite qui représente les propriétés des termes et sous-termes suite à l'exécution de l'atome, de la clause ou de la procédure. Les variables sur lesquelles elle est définie contient ou est égal à l'ensemble des variables $X_1,...,X_n$ puisque β_{in} est définie sur les variables de tête d'une clause et β_{out} sur toutes les variables de cette clause.

E_{ref_out} est une relation sur les tailles des termes et sous-termes qui définit les relations entre la taille des (sous-)termes concrets utilisés comme paramètres de l'exécution et les tailles des (sous-)termes concrets retournés par cette exécution.

E_{sol} est une relation sur les tailles qui exprime le nombre de solutions retournées par l'exécution en fonction de la taille des termes utilisés comme paramètres de cette exécution. Le nombre de solutions est représenté par la variable *sol*.

La sémantique d'une pseudo séquence abstraite B est donnée par la fonction de concrétisation Cc(B) :

(1) Si B = $\perp$ alors Cc(B)=$\varnothing$
Sinon

---

[4] Nous verrons ce que représente cette fonction de concrétisation de substitution abstraite un peu plus loin.
[5] Et rien qu'elles.

(2) $Cc(B) = \{\langle\theta,S\rangle \mid \theta \in Cc(\beta_{in})$,
S est une suite de substitutions concrètes,
(3) $Subst(S) \subseteq Cc(\beta_{out})$, chaque substitution concrète de S est une instance de β_{out},
(4) $(S \neq \langle\rangle \Rightarrow \theta \in Cc(\beta_{ref}))$,
(5) $(\theta' \in Subst(S) \wedge \langle t_i\rangle_{i\in I_{ref}} = DECOMP(\theta,\beta_{ref}) \wedge \langle s_i\rangle_{i\in I_{out}} = DECOMP(\theta',\beta_{out})$
$\Rightarrow (||t_i||)_{i \in I_{ref}} + (||s_i||)_{i \in I_{out}} \in Cc(E_{ref\_out}))$,
$(\langle t_i\rangle_{i\in I_{ref}} = DECOMP(\theta,\beta_{ref}) \Rightarrow (||t_i||)_{i\in I_{ref}} + \{sol \rightarrow |S|\} \in Cc(E_{sol}))\}$

Si la séquence abstraite B est égale à $\perp$, alors la concrétisation de B ne génère aucune séquence concrète. Autrement dit, aucune séquence concrète ne satisfait les propriétés définies par B.

(4) Supposons que θ soit généré par β_{in}. Si l'exécution à partir de θ retourne une liste S vide[6], cela signifie que l'exécution échoue. Dans ce cas, θ ne peut pas être générée par β_{ref}. Mais si S n'est pas vide, cela signifie qu'on est sûr que l'exécution est un succès (au moins une fois) quant elle débute avec θ. Donc θ doit être générée par β_{ref}.

La partie (5) de cette fonction de concrétisation tient compte du fait qu'une séquence abstraite doit pouvoir donner de l'information non seulement sur les termes associés aux variables $X_1,...,X_n$ dans β_{ref} et aux variables $X_1,...,X_m$ dans β_{out}, mais aussi sur tous les sous-termes qui composent ceux-ci. Or, au niveau abstrait, β_{ref} et β_{out} peuvent, par exemple, affirmer qu'un terme associé à X_i est une liste, et donc générer tous les termes qui sont des listes. Or une liste peut avoir un nombre quelconque de termes composants (par exemple X_i/[a,b] et X_i/[a,f(b,X),g(d),b]). Il est donc impossible pour une substitution abstraite de donner de l'information sur *tous* les sous-termes *concrets* qu'elle peut générer. Pour obtenir les informations sur un sous terme, on utilise l'opération DECOMP.

Prenons par exemple la substitution abstraite β_{out} ={X_1/forme d'une liste}dont la concrétisation va retourner toutes les substitutions concrètes $\theta_1,\theta_2,...$ où X_1 est associé à une liste ou à un terme instanciable à une liste. On a entre autres θ_1={X_1/[]}, θ_2={X_1/[f(a)], θ_3={X_1/[f(g(X))|f(a)]}. On voit que la seule information donnée sur X_1 par β_{out} ne suffit pas pour donner les propriétés des termes concrets f(a), a, g(X) et X. De plus, on ignore si la concrétisation de β_{out} génère ces termes.

L'utilisation de DECOMP à partir de β_{out} et d'une substitution concrète générée par β_{out}, par exemple θ_3, va retourner une suite de tuples de termes concrets tels que chaque tuple va décrire les termes *et sous-termes concrets qui sont générés par* β_{out}. DECOMP(θ_3,β_{out}) va retourner, entre autres, le tuple $\langle$[f(g(X))|f(a)], *g(X), X, a*$\rangle$ grâce auxquels on pourra affirmer que les termes g(X),X et a sont bien générés par β_{out}.

---

[6] La liste vide $\langle\rangle$ a une signification différente de la liste $\langle\perp\rangle$. Dans le premier cas, l'exécution est un échec *et se termine*, dans le second, l'exécution ne se termine pas (on recherche infiniment une solution).

Une séquence abstraite B est une pseudo séquence abstraite $\langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$ qui vérifie les propriétés suivantes :

- $\beta_{in} \neq \bot$;
- $\beta_{ref} \leq \beta_{in}$;
- Si β_{ref}, β_{out}, E_{ref_out} ou $E_{sol} = \bot$ alors $\beta_{ref} = \beta_{out} = E_{ref\_out} = E_{sol} = \bot$;
- $\forall \sigma \in Cc(\beta_{out}), \exists \theta \in Cc(\beta_{ref}) : \sigma_{|dom(\beta_{ref})} \leq \theta$.

La première de ces conditions veut que la concrétisation de toute substitution abstraite β_{in} génère au moins une substitution concrète. La seconde condition s'assure que les substitutions concrètes générées par la $Cc(\beta_{ref})$ constitue bien un sous-ensemble (éventuellement équivalent) des substitutions concrètes générées par $Cc(\beta_{in})$. Si β_{ref} définit les propriétés des substitutions concrètes pour lesquelles on est certain d'obtenir au moins un résultat, il est normal que toutes ces substitutions concrètes soient représentées par β_{in}. D'autres part, certaines substitutions concrètes générées par β_{in} qui sont utilisées pour exécuter un atome, une clause ou une procédure peuvent faire échouer cette exécution. Elles ne doivent donc pas être représentées par β_{ref}. Donc, $Cc(\beta_{in})$ génère un ensemble de substitutions concrètes qui contient ou est égal à celui généré par $Cc(\beta_{ref})$. On note cette propriété $\beta_{ref} \leq \beta_{in}$ [7].

La troisième condition nous dit que si aucune substitution concrète résultat n'est générée par $Cc(\beta_{out})$ ($\beta_{out}=\bot$ [8]), alors il n'y a aucune substitution concrète en entrée pour laquelle on soit sûr d'obtenir un résultat (et donc $\beta_{ref}=\bot$). Si aucune substitution concrète n'existe suite à $Cc(\beta_{ref})$ et $Cc(\beta_{out})$, alors il est impossible de définir des relations sur les tailles de termes qui n'existent pas, et donc E_{ref_out} et $E_{sol} = \bot$ [9].

Enfin, la quatrième condition nous assure que toutes les substitutions concrètes générées par β_{out} sont bien des instanciations d'une substitution concrète générée par $Cc(\beta_{ref})$. En effet, il est normal que les substitutions concrètes générées par β_{out} soient l'instanciation d'une substitution concrète générée par β_{ref}.

La sémantique d'une séquence abstraite est identique à celle d'une pseudo séquence abstraite.

8.2 COMPORTEMENT

Le comportement abstrait d'une procédure est une formalisation des spécifications du « comportement de la procédure » fournie par l'utilisateur. Celui-ci

---

[7] $\beta_1 \leq \beta_2$ signifie que la concrétisation de β_1 génère un ensemble de substitutions concrètes inclus dans celui généré par la concrétisation de β_2. On dira aussi que β_1 est plus précis (ou encore, est une instance) que β_2 et que β_2 est plus général que β_1.

[8] $\beta=\bot$ signifie, comme nous le verrons pour les substitutions abstraites, que la concrétisation de la substitution β ne génère aucune substitution concrète.

[9] La notation $E=\bot$ signifie qu'il n'y a pas de termes sur lesquels on peut définir des relations.

peut, grâce au langage de spécification, identifier chaque procédure et y associer un ensemble de propriétés qui définissent sont comportement.

On choisit de représenter un comportement par :

- les informations sur les substitutions en entrée de la procédure par une *substitution abstraite* ;
- les informations sur les substitutions qui doivent mener à au moins un succès si elles sont utilisées en entrée pour exécuter une procédure par *une substitution abstraite* ;
- les informations sur les substitutions en sortie de la procédure *par une substitution abstraite* ;
- les informations sur les relations entre les tailles des termes présents dans les substitutions en entrée qui doivent mener à un succès, et celles des termes des substitutions en sortie *par une relation sur les tailles* ;
- les informations sur les relations entre la taille des termes présents dans la substitution en sortie d'une procédure et le nombre de solutions de cette procédure *par une relation sur les tailles* ;

On le voit, cette représentation correspond à la définition donnée pour les séquences abstraites. Notons que la représentation d'un comportement à l'aide de séquences abstraites est liée au langage de spécification des comportements. Dans le cas de l'analyseur, cette réutilisation va nous permettre de comparer sur base d'une même structure les résultats calculés par une procédure et l'ensemble de ses comportements.

Un comportement abstrait Beh_p pour une procédure p d'arité n a la forme $\langle p/n, \{\langle B_1,se_1\rangle, ..., \langle B_m,se_m\rangle\}\rangle$ où :

- $B_1,...,B_m$ sont des séquences abstraites telles que $dom_{in}(B_k) = dom_{out}(B_k) = \{X_1,...,X_n\}$;
- $se_1,...,se_m$ sont des expressions linéaires positives[10] définies sur $X_1,...,X_n$.

Chaque séquence B_i définit une classe d'appels en entrée et l'ensemble des propriétés des substitutions résultats de l'exécution de p à partir de cette classe d'appels. Se_i définit la méthode de calcul de la taille des termes à appliquer sur les termes de B_i.

$Sbeh = (Beh_p)p \in P$ est l'ensemble fini des comportements abstraits pour le programme P contenant exactement un comportement Beh_p pour chaque procédure p présente dans P. On suppose que l'information donnée par Sbeh est correct pour tous les appels qui peuvent survenir lors de l'exécution de P.

Remarque : β_{ref} ne doit donc pas ajouter de l'information sur les termes associés aux variables, mais spécialiser cette information. Par exemple, on a une procédure p qui utilise deux arguments X et Y. On suppose que pour n'importe quelle

[10] Ces expressions son définies dans l'annexe consacrée aux notions propres à l'implantation.

utilisation de p, X est toujours une variable différente de Y. Cette information doit être représentée dans β_{in} *et* β_{ref}, et pas seulement dans β_{ref}. En effet, l'analyseur est incapable de déterminer que la procédure a le bon comportement si $X \neq Y$, il peut seulement vérifier si la procédure se comporte comme prévu quand X est différent de Y.

8.3 SUBSTITUTION ABSTRAITE

Une substitution abstraite définit les propriétés d'un ensemble de substitutions concrètes de la forme $\{X_1/t_1,...,X_n/t_n\}$.

Au niveau abstrait, on associe un index unique à chaque variable $X_1,...,X_n$. Cet index identifie une variable et les informations associées à celle-ci. Il est important de comprendre cette notion d'index car il est la base de la manipulation des substitutions abstraites. Les informations associées à chaque index sont le mode, le type, la forme et les partages de variables.

Mais, dans une substitution concrète, une variable peut être associée à un terme composé de sous-termes. Si on se limite à associer les index aux variables $X_1,...,X_n$ seulement, on ne sait pas représenter d'informations sur ces sous-termes[11]. Or, on veut pouvoir définir le même type d'information sur chacun d'eux, tout comme on le fait pour les variables. Pour cela, une substitution abstraite va aussi manipuler des index qui ne sont pas associés directement à des variables mais qui représentent chacun un sous-terme et auxquels on va associer les propriétés de ce sous-terme.

La définition d'une substitution abstraite procède en plusieurs étapes : on définit d'abord la notion de Atuple, ensuite celle de pseudo substitution abstraite, et enfin la substitution abstraite.

Atuple.

Pour tout ensemble d'index I, on décrit par *Atuples*$_I$ l'ensemble des 5-tuples $\langle mo,ty,frm,ps,E\rangle$ où $mo \in Modes_I$, $ty \in Types_I$, $frm \in Frames_I$, $ps \in PSharing_I$ et $E \in Sizes_I$. Chacun de ces composants représente un type de propriétés que l'on veut capturer sur les programmes analysés. Ils seront définis plus loin. Nous dirons seulement pour le moment que *mo* et *ty* donne respectivement le mode et le type associé à chaque index de I. Le composant *frm* associe une forme aux index pour lesquels le mode n'est pas « variable ». Le composant *ps* définit leurs partages de variables entre les différents index et le composant E donne la taille de chacun d'eux.

La sémantique d'un élément $\alpha \in Atuples_I$ est donnée par la fonction de concrétisation :

$$Cc(\alpha) = Cc(mo) \cap Cc(ty) \cap Cc(frm) \cap Cc(ps) \cap Cc(E)$$

[11] On ne peut associer de l'information qu'à un index.

La concrétisation d'un a-tuple α retourne l'ensemble des tuples de termes $\langle t_1,...,t_n\rangle$ ($I=\{1,...,n\}$) où chaque t_i satisfait les propriétés définies par mo, ty, frm, ps et E.

Une pseudo substitution abstraite β définie sur l'ensemble des index I est soit $\perp$, soit un tuple de la forme $\langle sv,\alpha\rangle$ où le composant sv est une fonction totale sv : $(dom(\beta)=\{X_1,...,X_n\})\rightarrow I$ et $\alpha=\langle mo,ty,frm,ps,E\rangle \in Atuple_I$ avec mo, ty, frm, ps, $E \neq \perp$.

Le composant sv établit la correspondance entre chaque variable et l'index qui y est associé. Le composant mo (ty) donne le mode (type) de chaque terme défini.

La sémantique d'une substitution abstraite est donnée par la fonction de concrétisation $Cc(\beta)$:

$$\text{Si } \beta = \perp \text{ alors } Cc(\beta) = \varnothing$$
$$\text{sinon } Cc(\beta) = \{\theta \mid dom(\theta)=dom(\beta) \wedge \exists \langle t_i\rangle_{i \in I} \in Cc(\alpha) :$$
$$(\forall X \in dom(\beta) \wedge X\theta = t_{sv(X)}),$$
$$(\forall i \in I : frm(i)=f(i_1,...,i_n) \Rightarrow t_i=f(t_{i_1},...,t_{i_n})), (\langle t_i\rangle_{i \in I} \in Cc(\alpha))\}.$$

Si la pseudo substitution abstraite est $\perp$, alors elle ne génère aucune substitution concrète puisqu'aucun terme pouvant être associé aux variables $X_1,...,X_n$ ne satisfait toutes les propriétés définies par le a-tuple α de β.

Si la pseudo substitution abstraite est différente de $\perp$, alors sa concrétisation génère toutes les susbtitutions concrètes de domaine $\{X_1,...,X_n\}$ et où les termes associés à chaque variable satisfont toutes les propriétés définies par le a-tuple α de β. Si ces termes sont composés, on est certain que tous leurs sous-termes répondent eux aussi à ces propriétés.

Remarque : rien ne lie le nombre d'indices[12] utilisés dans une pseudo substitution abstraite et les variables $X_1,...,X_n$ de celle-ci. En effet, le nombre d'indices peut être supérieur à n parce que la forme de certains indices peut être développée. Dans ce cas, il existe un ou plusieurs indices qui définissent cette forme développée et qui ne sont pas associés à une variable et. D'autre part, le nombre d'indices peut être inférieur à n puisque plusieurs variables peuvent être associées au même indice[13].

Une substitution abstraite β définie sur I est une pseudo substitution abstraite. Si elle est différente de $\perp$, elle doit vérifier les deux conditions suivantes :

- le composant frm ne possède pas de circuits (cette condition est assuré par le composant frm lui-même) ;
- tous les indices utilisés sont soit associés à une variable soit utilisés dans une forme quelconque.

La sémantique d'une substitution abstraite est équivalente à la sémantique de la pseudo substitution abstraite.

---

[12] Nous utilisons indifféremment le terme *indice* et *index*.
[13] C'est le moyen retenu pour représenter le partage de valeurs.

8.4 LE COMPOSANT SV

Sv est une fonction surjective définie sur un ensemble de variables $D=\{X_1,...,X_n\}$ et qui associe a chacune d'elles un index i appartenant à un ensemble d'index I.

$sv : D \rightarrow I$

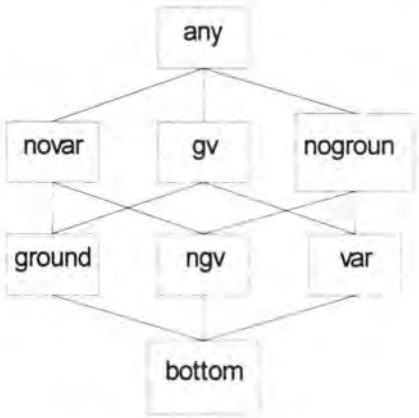
Plusieurs variables peuvent être associées au même index. La sémantique de ce composant est donnée par la fonction de concrétisation Cc(sv) :

$$Cc(sv) = \{\theta \mid dom(\theta)=D \wedge \forall X_i, X_j \in D : sv(X_i)=sv(X_j) \Rightarrow X_i\theta=X_j\theta\}$$

La concrétisation d'une fonction sv génère l'ensemble des substitutions concrètes définies sur les variables $X_1,...,X_n$ auxquelles sont associées des termes quelconques. Si deux variables sont associées au même indice, alors elles sont associés au même terme dans chacune des substitutions concrètes.

8.5 LES MODES ET LE COMPOSANT MODE

Une première propriété calculée sur les programmes est le mode des termes. Pour représenter cette propriété, on définit l'ensemble Modes = {⊥, ground, var, ngv, noground, novar, gv, ⊤} muni de la relation d'ordre définie par le diagramme suivant :



On dira que le mode i_1 est « supérieur » au mode i_2 ($i_1 > i_2$) si, dans le diagramme, i_1 est au dessus de i_2 et si un arc rejoint i_1 à i_2.

La sémantique des modes est donnée par la fonction de concrétisation :

$Cc(\bot)$	$= \varnothing$
Cc(ground)	= {t | t est un terme clos}
Cc(var)	= {t | t est une variable}
Cc(ngv)	= {t | t n'est ni un terme clos ni une variable}
$Cc(LUB_{mode}(M_1,M_2))$	$= Cc(M_1) \cup Cc(M_2)$

La concrétisation de la valeur *ground, var* et *ngv* génère respectivement l'ensemble des termes clos, l'ensemble des termes qui sont des variables et l'ensemble

des termes qui ne sont ni clos ni des variables. La concrétisation des autres valeurs de Modes correspond à l'union de la concrétisation de valeurs plus élémentaires.

Pour tout ensemble d'index I, on note $Modes_I$ l'ensemble de toutes les fonctions de I sur Modes augmenté de $\perp$. La sémantique d'un élément mo $\in Modes_I$ est donnée par la fonction de concrétisation Cc(mo) :

Si mo = $\perp$ alors Cc(mo) = $\varnothing$
Sinon Cc(mo) = $\{\langle t_i \rangle_{i \in I} \in T^I \mid \forall i \in I : t_i \in Cc(mo(i))\}$

où T^I est l'ensemble des tuples de termes indicés par I. Si mo est $\perp$, sa concrétisation ne génère aucun terme concret. Sinon, elle génère un ensemble de tuples de termes indicés par I où chaque terme t_i satisfait le mode associé à i.

8.6 LES TYPES ET LE COMPOSANT TYPE

Une autre propriété calculée sur les programmes est le type des termes. Pour cela, on définit l'ensemble Types = {$\perp$, list, anylist, any} muni d'une relation d'ordre définie par :

$\perp \leq$ list $\leq$ anylist $\leq$ any.

La sémantique de l'ensemble Types est donnée par la fonction de concrétisation :

Cc($\perp$) = $\varnothing$
Cc(list) = {t | t est une liste}
Cc(anylist} = {t | t est un terme qui peut être instancié à une liste}
Cc(any) = {t | t es un terme quelconque}

On remarquera que toute terme qui est une variable X appartient à Cc(anylist), et que tout terme qui peut être instancié à une liste (par exemple [a|X]) appartient aussi à Cc(anylist).

Pour tout ensemble d'index I, on note $Types_I$ l'ensemble de toutes les fonctions de I sur Types augmenté de $\perp$. La sémantique d'un élément ty $\in Types_I$ est donnée par la fonction de concrétisation Cc(ty) :

Si ty = $\perp$ alors Cc(ty) = $\varnothing$
Sinon Cc(ty) = $\{\langle t_i \rangle_{i \in I} \in T^I \mid \forall i \in I : t_i \in Cc(ty(i))\}$.

où T^I est l'ensemble des tuples de termes indicés par I. Si ty est $\perp$, sa concrétisation ne génère aucun terme concret. Sinon, elle génère un ensemble de tuples de termes indicés par I où chaque terme t_i satisfait le type associé à i.

8.7 FORMES ET COMPOSANT FRM

On définit tout d'abord F_p l'ensemble de toutes les formes possibles définies sur l'ensemble d'index $I_p=\{1,...,p\}$. Ces formes sont représentées par des expressions $f(i_1,...,i_q)$ où f est un symbole de fonction d'arité q et $i_1,...,i_q \in I$.

On y retrouve pas exemple f(1,3,4), f(1,2,4), g(5), [1|3][14],etc.

Le composant frm est la fonction partielle *frm* : $I_p \rightarrow F_p$ qui associe à l'index i une forme appartenant à F_p et qui ne contient pas l'index i.

Par exemple, l'association de 2 à f(3,4) peut être définie dans frm. Par contre, l'association de 2 à f(2,3) n'est pas valide parce que la concrétisation des informations associées à 2 ne peut pas générer des termes qui se composent eux-mêmes[15].

On désigne par frm(i) = undef les indices pour lesquels on n'a pas de forme associée. On note Leaves(frm) = $\{i \in I \mid frm(i)=undef\}$.

Pour tout ensemble d'index I, on note $Pattern_I$ l'ensemble de toutes les fonctions partielles de I sur T_I^* [16]. La sémantique d'un élément frm $\in$ $Pattern_I$ est donnée par la fonction de concrétisation Cc(frm) :

$$Cc(frm) = \{\langle t_i \rangle_{i \in I} \in T_I^* \mid \forall\, i,i_1,...,i_n \in I : frm(i)=f(i_1,...,i_n) \Rightarrow t_i = f(t_{i_1},...,t_{i_n})\}.$$

Cette concrétisation génère tous les tuples des termes composés indicés par I et des sous-termes qui composent ces premiers.

Par exemple, la concrétisation de frm = {1 →f(2,3), 2 →g(3)} (I={1,2,3}) génère entre autres les tuples ⟨f(g(a),a), g(a)⟩, ⟨f(g(Y),Y),g(Y)⟩.

8.8 PARTAGES DE VARIABLES ET COMPOSANT PS

Le partage de variables précise les index qui sont liés. Deux index sont liés si toute modification sur les informations associées à un index particulier se fait aussi sur les informations associées à l'autre index. Seuls les index dont le mode est variable peuvent partager des variables.

Le composant Ps est une relation binaire et symétrique sur les index {1,...,p} et exprime les partages de variables possibles entre deux index.

Exemple : on a les termes t_1=X et t_2=Y. Toute variable « partage avec elle-même ». Le ps contient alors {(1,1),(2,2)}. Mais si à un moment donné, les deux termes t_1 et t_2 sont unifiés, les variables qu'ils représentent sont désormais liées. Le ps devient alors {(1,1),(2,2),(1,2),(2,1)}.

---

[14] [1|3] est similaire à la notation .(1,3) où '.' est le symbole de fonction d'arité 2.

[15] Il serait possible d'avoir cette propriété si on tenait compte des termes infinis, ce qui n'est pas le cas.

[16] T_I^* est l'ensemble de tous tuples de les termes composés indicés par I.

Pour tout ensemble d'indices I, on note $PSharing_I$ l'ensemble de toutes les relations binaires et symétriques $ps \subseteq I \times I$ augmenté de $\perp$. La sémantique d'un élément $ps \in PSharing_I$ est donné par la fonction de concrétisation Cc(ps) :

$$\text{Si } ps = \perp \text{ alors } Cc(ps) = \varnothing$$
$$\text{Sinon } Cc(ps) = \{\langle t_i \rangle_{i \in I} \in T_I \mid \forall\, i,j \in I : Var(t_i) \cap Var(t_j) \neq \varnothing \Rightarrow (i,j) \in ps\}.$$

Si ps est $\perp$ alors sa concrétisation ne retourne aucun terme. Sinon, elle retourne un ensemble de tuples de termes (un terme pour chaque indice de I) qui partagent au moins une variable.

Le partage de variables d'index associés à une forme n'est pas repris dans le composant ps, mais il peut être recalculé grâce au composant frm. Ce calcul nous donne un composant ps portant sur *tous* les termes et que l'on note ps*. Il est la plus petite relation sur {1,...,p} satisfaisant les deux règles :

$\forall$ i,j,k $\in$ {1,...,p}:
i) $ps(i,j) \Rightarrow ps^*(i,j)$
ii) $frm(k)=f(...,j,...) \wedge ps^*(i,j) \Rightarrow ps^*(k,i)$

8.9 RELATION SUR LES TAILLES

Ce composant donne de l'information sur les relations entre les tailles des termes par rapport à une mesure donnée (ou norme) $\|.\| : T \rightarrow \aleph$.

Pour tout ensemble d'index I, on définit le domaine abstrait $Sizes_I$ dont les éléments représentent des tuples $\langle n_i \rangle_{i \in I}$ d'entiers naturels. Chacun de ces entiers représente la taille du terme t_i associé. On note E un élément de $Sizes_I$. Dans le cadre de l'analyseur, un élément E est un polyèdre de dimension I représenté par un système d'équations et d'inéquations linéaires[17] définis sur $Exp_{\{sz(i)\}_{i \in I}}$. On note $\perp$ le polyèdre vide. La sémantique des tailles est donnée par la fonction de concrétisation :

$$Cc(E) = \{\langle n_i \rangle_{i \in I} \in \aleph^{|I|} \mid (n_i)_{i \in I} \text{ est une solution de E}\}.$$

Un élément E de $Sizes_I$ représente aussi un ensemble de tuples de termes $\langle t_i \rangle_{i \in I}$. Soit T_E le même ensemble que E quand on est intéressé par les tuples de termes qu'il représente. La sémantique est alors donnée par la fonction de concrétisation :

$$Cc(T_E) = \{\langle t_i \rangle_{i \in I} \in T^I \mid (\|t_i\|)_{i \in I} \text{ est une solution de } T_E\}.$$

Note : un système d'équations linéaires sur $EXP_{\{sz(i)\}_{i \in I}}$ peut toujours être représenté par A.SZ=b, où A est une matrice nX|I|, SZ un vecteur colonne de variables $\{sz(i)\}_{i \in I}$ et b un vecteur colonne de n entiers naturels. De même, un système d'inéquations linéaires peut être représenté par C.SZ $\geq$ d, où C est une matrice mX|I|, SZ un vecteur colonne de variables $\{sz(i)\}_{i \in I}$ et d un vecteur colonne de m entiers naturels.

---

[17] Ce système d'équations sera pris en charge par la bibliothèque de polyèdre définie par [10].

On écrit les (in)équations entre double crochets [[...]] pour les exprimer sous leur forme syntaxique et non sous leur aspect « relation sémantique ». Si f est une fonction de l'ensemble des indices sur un autre ensemble d'indices telle que f(i)=i' et f(j)=j', l'expression [[sz(f(i)) = sz(f(j))+1]] doit être lue comme l'équation syntaxique sz(i')=sz(j')+1.

9. Les opérations abstraites

Les opérations abstraites redéfinissent la manière dont on va calculer les séquences abstraites tout en conservant la sémantique calculée par Prolog. Elles ne peuvent être définies de manière pratique qu'une fois le domaine abstrait choisi.

Le domaine retenu étant composé des modes, types, formes, partages de variables et de valeurs, relations sur les tailles des termes d'une même substitution et entre substitutions, relations entre les tailles des termes et le nombre de solutions, et des séquences et substitutions abstraites, on va définir un ensemble d'opérations abstraites portant chacune sur un ou plusieurs types d'information en même temps. Certaines d'entre elles prennent en charge un calcul très précis sur un type de propriété donné en tenant compte de résultats déjà calculés sur d'autres types de propriété. Ce sera par exemple le cas des opérations qui calculent les types des termes. La définition que nous donnerons de ces opérations sera, comme pour l'algorithme abstrait[1], limitée à la présentation des spécifications. Ces spécifications définissent les propriétés que toute implantation doit respecter. Il existe plusieurs implantations possibles pour une même spécification. Cependant certaines seront plus précises que d'autres. On retiendra bien sûr les implantations qui calculent au mieux les propriétés d'un programme. Ces implantations sont données dans l'annexe « Implantation des opérations abstraites ». Il s'agit encore une fois d'implantations générales.

[1] A la différence qu'ici le domaine abstrait est connu.

Les opérations qui calculent les modes, formes, partages de variables et de valeurs et les substitutions abstraites étant reprises (et éventuellement réadaptées) du système GAIA, nous les développerons moins que celles qui calculent les types et les séquences abstraites. Quant aux relations sur les tailles, comme elles ont peu été utilisées pendant la réalisation de ce travail, nous n'en donnerons que quelques spécifications, comme par exemple, les opérations de raffinement.

Encore une fois, les spécifications de ces opérations se base principalement sur les fonctions de concrétisations.

9.1 OPÉRATIONS SUR LES SÉQUENCES ABSTRAITES

Une remarque doit être faite pour toutes les opérations qui calculent les séquences abstraites : comme les composants d'une même séquence évoluent indépendamment les uns des autres, il est souvent nécessaire de rétablir la correspondance entre les indices utilisés dans chacun d'eux. Cette correspondance est établie à l'aide des « structural mappings » et des « constrained mappings ». Ces deux notions sont plus détaillées dans l'annexe consacrée aux notions propres à l'implantation. Nous essayerons de limiter leur utilisation dans les spécifications qui suivent et, si cette utilisation est nécessaire, nous signalerons simplement qu'une correspondance doit être établie. Nous noterons alors ces deux notions *tr*.

9.1.1 Préordre sur les séquences abstraites

$B_1 \leq B_2$

La relation $\leq$ définit un préordre sur les séquences abstraites qui pourra être utilisé (entre autres) pour comparer le résultat d'une procédure au comportement de cette procédure. Elle s'établit sur base des relations de préordre de chacun des composants des séquences abstraites. Ainsi, B_1 est inférieur ou égal à B_2 (on dira aussi que B_1 est plus précis (ou est une instance de) que B_2) si les conditions suivantes sont respectées :

- les substitutions abstraites en entrée des deux séquences sont égales (elles sont définies sur le même ensemble de variables et elles génèrent le même ensemble de substitutions concrètes) ;
- la concrétisation de la substitution abstraite raffinée de B_1 génère un ensemble de substitutions concrètes inclus ou égal à celui généré la substitution raffinée de B_2. ($\beta_{ref,1} \leq \beta_{ref,2}$ où la relation $\leq$ est le préodre défini sur les susbtitutions abstraites) ;
- la concrétisation de la substitution abstraite en sortie de B_1 génère un ensemble de substitutions concrètes inclus ou égal à celui généré la substitution abstraite en sortie de B_2. ($\beta_{out,1} \leq \beta_{out,2}$ où la relation $\leq$ est le préodre défini sur les susbtitutions abstraites) ;

- Une fois les correspondances établies entre les indices des substitutions abstraite β_{ref} de B_1 et β_{ref} de B_2 et entre β_{out} de B_1 et β_{out} de B_2, le polyèdre représenté par l'expression sur les tailles de B_1 doit être inclus ou égal à celui représenté par l'expression sur les tailles de B_2 ;

- Une fois la correspondance établie entre les indices des substitutions abstraite β_{ref} de B_1 et β_{ref} de B_2 ainsi que celle entre la variable sol de B_1 et la variable sol de B_2, le polyèdre représenté par la relation entre les tailles des termes et le nombre de solutions de B_1 doit être inclus ou égal à celui représenté par la même relation dans B_2.

$$
\begin{aligned}
B_1 \leq B_2 \Leftrightarrow \quad & \beta_{ref,1} \leq \beta_{ref,2} \qquad (\exists\, tr_{ref} : I_{ref,2} \rightarrow I_{ref,1}) \\
& \beta_{out,1} \leq \beta_{out,2} \qquad (\exists\, tr_{out} : I_{out,2},\, I_{out,1}) \\
& (tr_{ref} + tr_{out})^{<}(E_{ref\_out,1}) \leq E_{ref\_out,2} \\
& (tr_{ref} + \{sol \rightarrow sol\})^{<}(E_{sol,1}) \leq E_{sol,2}
\end{aligned}
$$

9.1.2 Normalisation d'une séquence abstraite

NORMALIZE(B) = B'.

Cette opération transforme la pseudo séquence abstraite B en une séquence abstraite B'.

Certaines opérations retournent des pseudo séquences abstraites, ce qui facilite la spécification de l'opération en question. Il est alors possible de transformer cette pseudo séquence abstraite en une séquence abstraite.

Cette opération respecte la propriété suivante :

$$
\begin{aligned}
\text{Si} \quad & \langle \theta, S \rangle \in Cc(B) \\
& \theta \in Cc(\beta_{ref}) \Rightarrow \theta \in Cc(\beta_{in}) \\
& \sigma \in Subst(S) \Rightarrow \sigma_{|dom(\beta_{ref})} \leq \theta \\
\text{Alors} \quad & \langle \theta, S \rangle \in Cc(B')
\end{aligned}
$$

La concrétisation de B génère un ensemble de séquences concrètes $\langle \theta, S \rangle$. Si la substitution θ est générée par la concrétisation de la substitution abstraite raffinée de B, alors elle doit aussi l'être par la substitution abstraite en entrée de B. De plus, la projection de chacune des substitutions σ résultant d'une exécution à partir de θ sur le domaine de θ est une instance de θ. Si une séquence satisfait ces deux conditions, elle doit être générée par B'.

9.1.3 Création d'une séquence abstraite initiale

Empty_SA(β) = B.

Cette opération initialise une séquence abstraite avec une substitution abstraite en entrée. Elle vérifie la propriété suivante :

$$\forall\, \theta \in Cc(\beta) : \langle \theta, \langle\rangle \rangle \in Cc(B)$$

9.1.4 Concaténation de séquences abstraites

$CONC\_SA(B_1, B_2) = B'$.

La concaténation de deux séquences abstraites B_1 et B_2 a pour but de créer une séquence abstraite B' qui représente en même temps toutes les propriétés définies par les deux séquences abstraites B_1 et B_2. Elle est utilisée quand on a calculé les séquences abstraites résultant de l'exécution de différentes clauses et que l'on veut représenter la séquence abstraite résultant de l'exécution de la procédure constituée de ces clauses.

L'opération respecte la propriété suivante :

$$\langle\theta,S_1\rangle \in Cc(B_1) \wedge \langle\theta,S_2\rangle \in Cc(B_2) \Rightarrow \langle\theta, S_1 : :S_2\rangle \in Cc(B').$$

$Cc(\beta_1)$ et $Cc(\beta_2)$ génèrent deux ensembles de séquences concrètes. Pour toutes les séquences concrètes de ces deux ensembles qui ont même substitution en entrée θ, on concatène leurs suites de substitutions réponses. On crée ainsi une séquence concrète unique. Cette séquence doit être générée par la concrétisation de B'. B' représente donc toutes les propriétés de toutes les substitutions résultats. La concaténation de deux séquences abstraites correspond à la plus petite borne supérieur de ces deux séquences.

Remarque : la concaténation de plusieurs séquences abstraites $B_1,...,B_n$ correspond à la concaténation $CONC\_SA(B_1, CONC\_SA(B_2,...,CONC\_SA(B_{n-1},B_n))...) = B'$.

9.1.5 Plus grande borne inférieure de deux séquences abstraites

$JOIN\_SA(B_1,B_2) = B'$.

Cette opération calcule la borne inférieure la plus grande de deux séquences abstraites B_1 et B_2.

Chacune des séquences abstraites B_1 et B_2 décrit un ensemble de propriétés. La plus grande borne inférieur de ces ensembles de propriétés correspond à leur intersection. B' ne décrit donc que les propriétés communes aux deux séquences abstraites B_1 et B_2.

L'opération respecte la condition suivante :

$$\langle\theta,S\rangle \in Cc(B_1) \wedge \langle\theta_{|dom_{in}(B2)}, S_{|dom_{out}(B2)}\rangle \in Cc(B_2) \Rightarrow \langle\theta,S\rangle \in Cc(B').$$

Toutes les séquences générées par la concrétisation de B' sont générées par la concrétisation de B_1 et, si elles sont restreintes aux domaines de définition des substitutions abstraites de B_2, sont aussi générées par la concrétisation de B_2. On voit qu'il est nécessaire que le domaine de définition des substitutions abstraites en entrée et en sortie de B_2 soit inclus dans le domaine de définition des substitutions abstraite en entrée et en sortie de B_1.

9.2 OPÉRATIONS SUR LES SUBSTITUTIONS ABSTRAITES

9.2.1 Normalisation d'une substitution

CLEANUP(β) = β'

Cette opération calcule une substitution abstraite à partir d'une pseudo substitution abstraite. Comme pour les séquences abstraites, certaines opérations sont plus facilement définies si elles manipulent des pseudo substitutions abstraites. Cette opération permet de transformer à tout moment en substitutions abstraites.

L'opération respecte la propriété suivante :

$\theta \in Cc(\beta) \wedge \theta$ ne contient pas de termes infinis[2] $\Rightarrow \theta \in Cc(\beta')$.

Toutes les substitutions concrètes générées par Cc(β) qui ne définissent pas de termes infinis sont générés par β'.

9.2.2 Plus petite borne supérieure de deux substitutions abstraites

LUB(β_1,β_2) = β = UNION(β_1,β_2).

Cette opération calcule la plus petite borne supérieure de deux substitutions abstraites β_1 et β_2. Elle calcule une substitution abstraite qui représente l'ensemble des propriétés représentées par β_1 et β_2. Elle est utilisée lorsque l'on veut calculer la substitution abstraite en sortie d'une procédure à partir des substitutions abstraites en sortie des clauses de cette procédure, et ainsi obtenir l'ensemble approximé des résultats retournés par la procédure.

L'opération doit respecter les propriétés suivantes :

i) $dom(\beta)=D \wedge Cc(\beta_1) \cup Cc(\beta_2) \subseteq Cc(\beta)$.
ii) $\beta_1, \beta_2 \leq \beta \wedge \forall \beta' : dom(\beta')=D : (\beta_1, \beta_2 \leq \beta') \Rightarrow (\beta \leq \beta')$

Puisque β est une substitution abstraite qui représente toutes les propriétés des deux substitutions abstraites β_1 et β_2, toute substitution concrète générée par la concrétisation d'une de ces deux substitutions doit aussi être générée par la concrétisation de β. D'autre part, si β est la plus petite borne supérieure de β_1 et β_2, alors elle est unique : la concrétisation de toute autre substitution abstraite β' définie sur le même domaine que β génère toutes les substitutions générées par Cc(β) (et donc au moins celles générées par Cc(β_1) et Cc(β_2)) mais en plus, elle génère au moins une substitution concrète qui n'est pas générée par β.

La précision de l'implantation de cette opération est importante : Cc(β) doit *au moins* générer les substitutions concrètes représentées par β_1 et β_2. Si cette opération

---

[2] Permet de ne considérer que des termes qui n'ont pas de circuits dans leur forme (un terme ayant un circuit étant un terme infini).

est pu précise, Cc(β) générera beaucoup de substitutions qui ne sont pas représentées par β_1 et β_2. Plus elle sera précise, moins Cc(β) générera de telles substitutions. On dira que β est l'union stricte de β_1 et β_2 si Cc(β) ne génère aucune autre substitution concrète que celles générées par $Cc(\beta_1)$ et $Cc(\beta_2)$.

Remarque : tout comme pour la plus petite borne supérieure des séquences abstraites, on peut définir la plus petite borne supérieure des substitutions abstraites $\beta_1,...,\beta_n$ comme étant : $LUB(\beta_1,...\beta_n) = LUB(...LUB(LUB(\perp, \beta_1), \beta_2)...,\beta_n)$.

9.2.3 Opération Ext_lub

$EXT\_LUB(\beta_1,\beta_2) = \langle \beta', st \rangle$.

Cette opération est une amélioration de l'opération LUB de substitutions abstraites. Elle calcule $\beta=LUB(\beta_1,\beta_2)$ mais en plus, elle indique à l'aide du booléen *st* si β est l'union stricte de β_1 et β_2 ($Cc(\beta')=Cc(\beta 1)\cup Cc(\beta 2)$).

9.2.4 Préordre sur les substitutions abstraites

$\beta_1 \leq \beta_2$.

Cette opération vérifie si l'ensemble des substitutions concrètes généré par $Cc(\beta_1)$ est inclus ou égal à celui généré par $Cc(\beta_2)$. Si c'est le cas, nous disons que β_1 est plus précis que β_2, ou encore que β_1 est une instance de β_2. Pour que cette comparaison puisse s'effectuer, les deux substitutions abstraites doivent être définies sur le même ensemble de variables $\{X_1,...,X_n\}$.

9.2.5 Plus grande borne inférieure de deux substitutions abstraites

$JOIN_s(\beta 1,\beta 2) = GLB_{subst}(\beta_1,\beta_2) = \beta'$

Cette opération calcule une substitution abstraite β' qui représente les propriétés communes aux deux substitutions abstraites β_1 et β_2. La concrétisation de β' génère toutes les substitutions abstraites qui sont générées par $Cc(\beta_1)$ et par $Cc(\beta_2)$. Cependant, si les domaines de définition de β_1 et β_2 sont équivalents, β' est lui aussi défini sur ce domaine. Si ce n'est pas le cas, les termes associés aux variables utilisées dans β_1 et dans β_2 sont toujours équivalents, et les termes associés aux variables propres à une substitution abstraire seulement satisfont les propriétés définies par cette substitution abstraite sur cette variable. Donc β' est définie sur l'union des domaines de β_1 et β_2.

L'opération satisfait la propriété :

$\theta_k \in Cc(\beta_k)\ (k=1,2) \wedge \theta_{1_{|D}} = \theta_{2_{|D}} \Rightarrow \theta_1 \circ \theta_2 \in Cc(\beta')$
où $D=dom(\beta_1) \cap dom(\beta_2)$.

L'opérateur ▫ est défini dans l'annexe consacrée aux notions propres à l'implantation. Appliqué aux susbtitutions concrètes, il consiste à retenir les X_i/t_i communes à θ_1 et θ_2 et les X_j/t_j propres à un seul θ_j.

9.2.6 Opération de raffinement Ref_{Ref}

$Ref_{Ref}(\beta_1,\beta_2) = \beta'$.

Cette opération est utilisée pour calculer une substitution abstraite raffinée β' sur base d'une substitution abstraite β_1 utilisée pour exécuter un atome, une clause ou une procédure et de la substitution abstraite β_2 résultant d'une exécution quelconque (pas nécessairement réalisée à partir de β_1).

β_1 définit les propriétés de toutes les substitutions concrètes θ_1 qui peuvent être utilisées pour exécuter un atome, une clause ou une procédure. La substitution β_2 définit quant à elle les propriétés des substitutions concrètes θ_2 qui résultent d'une exécution quelconque. Comme la substitution raffinée doit décrire les substitutions concrètes θ_1 à partir desquelles on est certain que l'exécution se termine au moins une fois, on recherche les θ_1 telles que θ_2 est une instance de θ_1. Une fois ces θ_1 trouvée, on en calcule les propriétés et celles-ci définissent β'.

Les domaines de β_1 et β_2 ne doivent pas obligatoirement être les mêmes.

L'opération respecte la propriété suivant :

$$\theta_k \in Cc(\beta_k)\ (k=1,2) \wedge \theta_{2\ |dom(\beta_1)} \leq \theta_1 \Rightarrow \theta_1 \in Cc(\beta').$$

Toute substitution concrète générée par $Cc(\beta_2)$ et projetée sur le domaine de β_1 est une instance d'une substitution concrète générée par $Cc(\beta')$, et toute substitution concrète générée par $Cc(\beta')$ est une substitution concrète générée par $Cc(\beta_1)$. $Cc(\beta_1)$ peut donc générer des substitutions qui feront échouer une exécution quelconque. Le domaine de définition de β' est équivalent à celui de β_1.

9.3 OPÉRATIONS DE RAFFINEMENT

Ces opérations calculent les polyèdres définissant les relations sur les tailles des termes entre eux, les relations sur les tailles des termes en entrée et ceux en sortie et les relations entre le nombre de solutions et la taille des paramètres suite aux exécutions des littéraux, clauses et procédures. En effet, on doit recalculer les relations pour tout ce qui touche au raffinement des termes en entrée et au nombre de solutions.

Comme les composants E n'ont pas beaucoup été considérés tout au long de ce travail, nous ne donnerons que les spécifications suivantes.

9.3.1 Opération Ref_{Ref_out}

$Ref_{Ref\_out}(\beta_1, \beta_1', E, \beta_2, \beta_2') = E'$.

Soient :
$\beta_k = \langle sv_k, \alpha_k \rangle$ des substitutions abstraites sur l_k (k=1,2) ;
$\beta_k' = \langle sv_k', \alpha_k' \rangle$ des substitutions abstraites sur l_k' (k=1,2) ;
$E \in Sizes_{l_1+l_2}$;
$E' \in Sizes_{l_1'+l_2'}$;
$dom(\beta_k)=dom(\beta_k')$ (k=1,2).

L'opération respecte la condition :

Si $\theta_k \in Cc(\beta_k) \cap Cc(\beta_k')$ (k=1,2)
$\langle t_{i,k} \rangle_{i \in l_k} = DECOMP(\theta_k, \beta_k)$ (k= 1,2)
$\langle u_{i,k} \rangle_{i \in l_k'} = DECOMP(\theta_k, \beta_k')$ (k=1,2)
$(||t_{i,1}||)_{i \in l_1} + (||t_{i,2}||)_{i \in l_2} \in Cc(E)$
Alors $(||u_{i,1}||)_{i \in l_1'} + (||u_{i,2}||)_{i \in l_2'} \in Cc(E')$

9.3.2 Opération $Refl_{Ref\_out}$

$Refl_{Ref\_out}(l, \beta_1, \beta_1', E, \beta_2, \beta_2') = E'$.

Soient :
l un littéral de la forme $X_{i_1}=X_{i_2}$ ou $X_{i_1}=f(X_{i_2},...,X_{i_n})$ ou $q(X_{i_1},...,X_{i_n})$;
$\beta_k = \langle sv_k, \alpha_k \rangle$ des substitutions abstraites sur l_k (k=1,2) ;
$\beta_k' = \langle sv_k', \alpha_k' \rangle$ des substitutions abstraites sur l_k' (k=1,2) ;
$E \in Sizes_{l_1+l_2}$;
$E' \in Sizes_{l_1'+l_2'}$;
$dom(\beta_k) = \{X_1,...,X_n\}$ (k=1,2) ;
$dom(\beta_k') = \{X_1,...,X_m\}$ (k=1,2) ($n \leq m$);
$\{X_{i_1},...,X_{i_n}\} \subseteq \{X_1,...,X_m\}$.

L'opération respecte la condition :

Si $\theta_k=\{X_1/t_{i_1},...,X_n/t_{i_n}\} \in Cc(\beta_k)$
$\theta_k'=\{X_1/t_1,...,X_m/t_m\} \in Cc(\beta_k')$
$\langle t_{i,k} \rangle_{i \in l_k} = DECOMP(\theta_k, \beta_k)$
$\langle u_{i,k} \rangle_{i \in l_k'} = DECOMP(\theta_k, \beta_k')$
$(||t_{i,1}||)_{i \in l_1} + (||t_{i,2}||)_{i \in l_2} \in Cc(E)$
Alors $(||u_{i,1}||)_{i \in l_1'} + (||u_{i,2}||)_{i \in l_2'} \in Cc(E')$

9.3.3 Opération Ref_{sol}

$Ref_{Sol}(\beta_1,\beta_2,E)=E'$.

Soient :
$\beta_1 = \langle sv_1, \alpha_1 \rangle$ une substitution abstraite sur l_1 ;
$\beta_2 = \langle sv_2, \alpha_2 \rangle$ une substitution abstraite sur l_2 ;
$E \in Sizes_{l_1+\{sol\}}$;

$E' \in Sizes_{I_1'+\{sol\}}$;
$dom(\beta_1)=dom(\beta_2)$.

L'opération respecte la condition :

Si $\theta \in Cc(\beta_1) \cap Cc(\beta_2)$
$\langle t_i \rangle_{i \in I_1} = DECOMP(\theta,\beta_1)$
$\langle u_i \rangle_{i \in I_2} = DECOMP(\theta,\beta_2)$
$(||t_i||) + \{sol \rightarrow s\} \in Cc(E)$
Alors $(||u_i||) + \{sol \rightarrow s\} \in Cc(E')$

9.3.4 Opération $Refl_{sol}$

$Refl_{sol}(l,\beta_1,\beta_2,E) = E'$.

Soient :
l un littéral de la forme $X_{i_1}=X_{i_2}$ ou $X_{i_1}=f(X_{i_2},...,X_{i_n})$ ou $q(X_{i_1},...,X_{i_n})$;
$\beta_1 = \langle sv_1, \alpha_1 \rangle$ définie sur I_1 ;
$\beta_2 = \langle sv_2, \alpha_2 \rangle$ définie sur I_2 ;
$E \in Sizes_{I_1+\{sol\}}$;
$E' \in Sizes_{I_1'+\{sol\}}$;
$dom(\beta_1) = \{X_1,...X_n\}$;
$dom(\beta_2) = \{X_1,...,X_m\}$ $(n \leq m)$;
$\{X_{i_1},...,X_{i_n}\} \subseteq \{X_1,...,X_m\}$

L'opération respecte la condition suivante :

Si $\theta_1=\{X_1/t_{i_1},...,X_n/t_{i_n}\} \in Cc(\beta_1)$
$\theta_2=\{X_1/t_1,...,X_m/t_m\} \in Cc(\beta_2)$
$\langle t_i \rangle_{i \in I_1} = DECOMP(\theta, \beta_1)$
$\langle u_i \rangle_{i \in I_2} = DECOMP(\theta,\beta_2)$
$(||t_i||) + \{sol \rightarrow s\} \in Cc(E_{sol})$
Alors $(||u_i||) + \{sol \rightarrow s\} \in Cc(E_{sol}')$

9.3.5 Opération $Combine_{out}$

$COMBINE_{out}(E_1,E_2)=E'$.

Soient :
$E_1 \in Sizes_{I_1 + I_2}$
$E_2 \in Sizes_{I_2 + I_3}$
$E' \in Sizes_{I_1 + I_3}$

L'opération respecte la condition :

Si $(n_{i,1})_{i \in I_1} + (n_{i,2})_{i \in I_2} \in Cc(E_1)$
$(n_{i,2})_{i \in I_2} + (n_{i,3})_{i \in I_3} \in Cc(E_2)$
Alors $(n_{i,1})_{i \in I_1} + (n_{i,3})_{i \in I_3} \in Cc(E')$

9.3.6 Opération $Combine_{sol}$

$COMBINE_{sol}(E,E_1,E_2)=E'$.

Soient :
$E \in Sizes_{I_1 + I_2}$;
$E_1 \in Sizes_{I_1 + \{sol\}}$
$E_2 \in Sizes_{I_2 + \{sol\}}$
$E' \in Sizes_{I_1 + \{sol\}}$

L'opération respecte la condition :

Si $(n_{i,1})_{i \in I_1} + (n_{i,2})_{i \in I_2} \in Cc(E_{ref\_out})$
$(n_{i,1})_{i \in I_1} + \{sol \rightarrow s_1\} \in Cc(E_{1,sol})$
$(n_{i,2})_{i \in I_2} + \{sol \rightarrow s_2\} \in Cc(E_{2,sol})$
Alors $(n_{i,1})_{i \in I_1} + \{sol \rightarrow s_1 * s_2\} \in Cc(E')$

9.3.7 Opération Compute_sz_const

Compute_sz_const(β_1, β_2) = E'.

Soient :
β_1 une subtitution abstraite sur I_1 ;
β_2 une substitution abstraite sur I_2 ;
$E \in Size_{I_1+I_2}$.

L'opération respecte la condition suivante :

Si $\theta_1 \in Cc(\beta_1)$
$\theta_2 \in Cc(\beta_2)$
$\theta_2 \leq \theta_1$
$\langle t_{i,1} \rangle_{i \in I_1} = DECOMP(\theta_1,\beta_1)$
$\langle t_{i,2} \rangle_{i \in I_2} = DECOMP(\theta_2,\beta_2)$
Alors $(||t_{i,1}||)_{i \in I_1} + (||t_{i,2}||)_{i \in I_2} \in Cc(E')$.

9.4 OPÉRATIONS SUR LES MODES

Les opérations sur les modes ont été définies dans le système GAIA. Nous n'en donnerons ici que les spécifications. Une présentation plus complète se trouve dans [9].

9.4.1 Construction de modes

$CONS_{mode}(f,M_1,...,M_n) = M'$. Cette opération calcule le mode d'un terme composé $f(t_1,...,t_n)$ à partir du mode des termes composant $f(t_1,...,t_n)$.

La construction de modes respecte la propriété :

$$\forall i : 1 \leq i \leq n : t_i \in CC(M_i) \Rightarrow f(t_1,...,t_n) \in Cc(M')$$

9.4.2 Extraction de modes

$EXTR_{mode}(M,f) = (M_1,...,M_n)$. Cette opération est l'inverse de la construction de modes. Elle calcule le mode le plus précis des termes $t_1,...,t_n$ de sorte que $f(t_1,...t_n)$ ait le mode M.

L'extraction de modes respecte la propriété :

$$f(t_1,...,t_n) \in Cc(M) \Rightarrow \forall i : 1 \leq i \leq n : t_i \in Cc(M_i)$$

9.4.3 Matching de modes

$MATCH_{mode}(M, f, M_1,..,M_n) = M'$. Cette opération recalcule le mode d'un terme composé $f(t_1,...,t_n)$ quand un de ses termes peut avoir été instancié. Elle est utilisée pendant les opérations d'unification abstraite.

Le matching de modes respecte la condition :

$$t \in Cc(M) \wedge \forall i : 1 \leq i \leq n : t_i \in Cc(M_i) \wedge \exists \sigma : t\sigma = f(t_1,...t_n) \Rightarrow f(t_1,...,t_n) \in Cc(M').$$

9.4.4 Unification abstraite de modes

$UAT(M_1,M_2) = M'$. Cette opération calcule le mode M' d'un terme t résultant de l'unification abstraite de deux termes t_1 et t_2 de mode M_1 et M_2.

L'opération respecte la condition :

$$t_1 \in Cc(M_1) \wedge t_2 \in Cc(M_2) \wedge \sigma \text{ est le mgu de } t_1 \text{ et } t_2 \Rightarrow t_1\sigma \in Cc(M').$$

9.4.5 Instanciation abstraite de modes

$IAT_{mode}(M)=M'$. Cette opération calcule le mode M' d'un terme dont le mode était M et qui a été arbitrairement instancié.

L'opération respecte la condition :

$$t \in Cc(M) \Rightarrow t\sigma \in (M')$$

9.4.6 Instantiation abstraite spécialisée de modes

$IAT2_{mode}(M_1,M_2)=M'$. Cette opération est un raffinement de l'opération précédente. Elle calcule le mode M' d'un terme dont le mode était M_1 et qui a été instancié par la substitution d'une variable à un terme dont le mode est M_2.

L'opération respecte la propriété :

$$t_1 \in Cc(M_1) \wedge t_2 \in Cc(M_2) \Rightarrow t_1\{y \leftarrow t_2\} \in Cc(M').$$

9.4.7 Opération $Unist_{mode}$

$UNIST_{mode}(M) = M'$. Cette opération calcule le mode M' qui approxime l'ensemble des termes qui peuvent être instanciés au terme $t \in Cc(M)$.

9.5 OPÉRATIONS SUR LES TYPES

Les opérations sur les types utilisent les résultats calculés par les opérations sur les modes.

9.5.1 Construction de types

$CONS_{type}(M_1,...,M_n,M',f,T_1,...,T_n) = T'$.

Cette opération calcule le type d'un terme composé $f(t_1,...,t_n)$ à partir des modes et des types de ses composants t_i et en tenant compte de son nouveau mode M'. Les termes $t_1,...,t_n$ ont respectivement le mode $M_1,...,M_n$.

L'opération respecte la condition suivante :

Si $t_i \in Cc(T_i)$, $i \in \{1,...,n\}$
$t_i \in Cc(M_i)$, $i \in \{1,...,n\}$
$f(t_1,...,t_n) \in Cc(M')$
Alors $f(t_1,...,t_n) \in Cc(T')$.

Par exemple, $t \equiv [a|W]$ de mode ngv et de type anylist et $W \equiv f(X)$ de mode ngv et de type any. Le mode et le type de a ne peuvent plus être changés. Mais on suppose que le terme associé à W devient f(a), le mode de W devient ground et le type reste any. On doit reconstruire le type de t (on a déjà calculé son mode M' qui est clos). W étant associé à un terme clos qui n'est pas une liste, le nouveau type de t devient any.

9.5.2 Extraction de types

$EXTR_{type}(T,f,M,M_1',...,M_n') = (T_1',...,T_n')$

Cette opération est l'inverse de $CONS_{type}$. Elle calcule l'approximation la plus précise des types $T_1',...,T_n'$ des termes $t_1,...,t_n$ qui composent $f(t_1,...,t_n)$ à partir du nouveau type T de $f(t_1,...,t_n)$ et en s'aidant des nouveaux modes des termes $t_1,...,t_n$.

L'opération respecte la propriété suivante :

Si $f(t_1,...,t_n) \in Cc(T)$
$f(t_1,...,t_n) \in Cc(M)$
$t_i \in Cc(M_i'), i \in \{1,...,n\}$
Alors $t_i \in Cc(T_i'), i \in \{1,...,n\}$.

Par exemple, on sait d'un terme que son mode est ngv et que son type est anylist. La meilleure approximation que l'on puisse donner sur ses termes composants en sachant que le premier est une variable et le second est un terme ngv est que le type de ces termes sont anylist. Ils peuvent en effet s'instancier à n'importe quel terme, le premier parce qu'il est une variable et le second parce qu'on ignore s'il sera instancié à une liste.

Un terme concret qui répond à ces propriétés est [f(X)|W] de mode ngv et de type anylist.

9.5.3 Instantiation abstraite de types

$IAT_{type}(T) = T'$.

Cette opération calcule le nouveau type T' d'un terme dont le type était T et qui à été arbitrairement instancié.

L'opération vérifie la condition suivante :

$t \in Cc(T) \Rightarrow T\sigma \in Cc(T')$.

Un terme t dont le type est liste ne peut qu'être instancié à une liste. Donc, T'=liste. Un terme t qui est de type anylist peut soit s'instancier à une liste soit à un terme instanciable à une liste soit à un terme quelconque. Son type T' devient alors any.

9.5.4 Matching de types

$MATCH_{type}(T,f,T_1,...,T_n,M') = T'$.

Cette opération est utilisée pendant l'unification abstraite pour recalculer le type du terme composé $f(t_1,...,t_n)$ quand certains de ses termes peuvent avoir été instanciés et dont le résultat est l'ensemble des termes $t_1',...,t_n'$ de type $T_1,...,T_n$. T est le type de $f(t_1,...,t_n)$ avant unification et M' est le nouveau mode de $f(t_1,...,t_n)$.

L'opération respecte la condition :

Si $t \in Cc(\beta)$
$t_i \in Cc(T_i), i \in \{1,...,n\}$

$\exists \sigma : t\sigma = f(t_1,...,t_n)$
Alors $f(t_1,...,t_n) \in Cc(T')$.

Par exemple, avant unification on a t ≡ [a,X,Y]) de mode ngv et de type anylist. L'unification associe t à [a,b,[a|b]] dont le mode est clos. Les types de a et b sont tous any. Le type de [a|b] est une liste. Le matching détermine que le type de t n'est plus anylist mais list.

9.5.5 L'unification abstraite de types

$UAT_{type}(M_1,M_2,M',T_1,T_2) = T'$.

Cette opération calcule le type du résultat de l'unification de deux termes t_1 et t_2 en tenant compte de leut mode M_1, M_2 et de leur type T_1,T_2 avant l'unification et en sachant que le mode résultant de cette unification est M'.

L'opération respecte la condition :

Si $t_1 \in Cc(M_1)$, $t_1 \in Cc(T_1)$
$t_2 \in Cc(M_2)$, $t_2 \in Cc(T_2)$
$\sigma \in mgu(t_1,t_2)$
$t_1\sigma \in Cc(M')$
Alors $t_1\sigma \in Cc(T')$.

Par exemple, $t_1 \equiv [a|X]$ de mode ngv et de type anylist, et $t_2 \equiv [a|[f(d)]]$ de mode ground et de type any. L'unification $t_1=t_2$ associe le terme $[a|f(d)]$ à t_1. Le mode de t_1 devient clos. Mais le type de t_1 doit être recalculé : le terme étant clos, il ne peut plus être instancié à une liste. Le type de t_1 devient donc any.

9.5.6 Instantiation abstraite spécialisée

$IAT2_{type}(M_1,M_2,M',T_1,T_2) = T'$.

Cette opération calcule le type T' d'un terme t_1 dont le type était T_1 et qui a été instancié par une variable associée au terme t_2 dont le type était T_2.

L'opération respecte la condition :

Si $t_1 \in Cc(M_1)$, $t_1 \in Cc(T_1)$
$t_2 \in Cc(M_2)$, $t_2 \in Cc(T_2)$
$t_1\{Y/t_2\} \in Cc(M')$
Alors $t_1\{Y/t_2\} \in Cc(T')$.

Si t_1 est un terme qui réunit les propriétés définies par M_1 et T_1, t_2 un terme qui réunit les propriétés définies par M_2 et T_2, et que t_1 peut avoir été instancié à t_2, suite à quoi, son mode est devenu M'. Dans ce cas, ce terme est de type T'.

Par exemple, t_1/[a|Y] a pour mode ngv et pour type anylist, t_2=b a pour mode ground et pour type any. L'instanciation $t_1\{Y/t_2\}$ retourne le nouveau terme [a|b] associé à t_1 et de mode ground. Le type de t_1 ne peut donc plus être anylist

puisque tous les termes sont clos. De plus, b n'est pas une liste, donc le nouveau type de t_1 est any.

9.5.7 Opération $Uninst_{type}$

$UNINST_{type}(T) = T'$.

Cette opération calcule le type T' qui approxime l'ensemble des termes qui peuvent être instanciés au terme $t \in Cc(T)$. Elle est utilisée pour déterminer les propriétés d'un terme dont l'instanciation a généré un terme de type T.

Par exemple, t est une liste après instanciation. Donc l'instanciation[3] *qui a mené à t devait unifier une liste à un terme t'. Et seuls les termes qui sont des variables ou des termes associables à une liste peuvent devenir une liste. Donc le type de t' était* anylist.

9.6 OPÉRATIONS SUR LE COMPOSANT E

9.6.1 Opération Sum_{Sol}

$SUM_{sol}(E_1,E_2) = E'$.

Cette opération exprime la longueur d'une séquence abstraite obtenue par concaténation de deux séquences abstraites B_1 et B_2 qui ont respectivement le composant E_1 et E_2 défini. Chacune de ces séquences renseigne sur le nombre de solutions concrètes qui sont retournées à partir de la substitution concrète en entrée.

La concrétisation de E_1 retourne tous les tuples $n_{i,1}$ qui expriment la longueur des termes et le nombre de solutions concrètes générés par $Cc(E_1)$. De même pour E_2. Tous les tuples générés par E_1 et E_2 qui sont équivalents sont définis par les propriétés de E'. E' est donc l'intersection des polyèdres définis par E_1 et E_2. On ne retient de B_1 et de B_2 que les termes générés de même longueur et les séquences de même longueur.

La propriété de cette opération est :

Si $\langle n_{i,k} \rangle_{i \in I+\{sol\}} \in Cc(E_k)$ $(k=1,2)$
$n_{i,1} = n_{i,2} = n_i \; \forall \; i \in I$
$n_{sol} = n_{sol,1} + n_{sol,2}$
Alors $\langle n_i \rangle_{i \in I+\{sol\}} \in Cc(E')$.

[3] Pour rappel, l'instanciation d'un terme t est l'application d'une substitution θ sur t que l'on note tθ.

10. Etat actuel de l'analyseur

Du point de vue des spécifications, l'analyseur prend en charge l'analyse du mode, du type, de la forme, des partages de variables et des partages de valeurs de chaque terme. On y définit les structures et les opérations qui constituent le programme.

Pour ce qui est de l'implantation, ces spécifications, proches des notations mathématiques, ont été traduites en langage C++. Les composants mo, ty et frm qui constituent les substitutions abstraites sont représentées par des objets simulant le comportement de fonctions. La relation ps est, quant à elle, représentée par un graphe où chaque sommet représente l'indice d'un terme. Le composant E représentant les relations qui existent entre les tailles de termes d'une même substitution abstraite trouve pour le moment sa place dans le programme, mais sa composition, et les opérations qui y sont associées, n'ont pas encore d'équivalents en C++. Nous savons cependant que nous nous baserons sur la bibliothèque de gestion de polyèdres définie par [10]. L'identification des variables et des termes se fait sur base de leurs indices. Il a également fallu définir des constrained mappings sur ces indices afin de pouvoir réaliser toutes les opérations qui nécéssitent une correspondance entre les termes.

Notons que les composants des substitutions abstraites agissent indépendamment les unes des autres: c'est à la substitution abstraite qui les contient de s'organiser pour qu'elles soient cohérentes entre elles.

Pour ce qui est des séquences abstraites, les substitutions abstraites en entrée, en sortie et plus raffinée que β_{in} sont représentées par les substitutions concrètes définies ci-dessus. Quant aux composants E_{ref_out} et E_{sol}, comme leur structure est équivalente au composant E des substitutions abstraites, ils doivent eux aussi encore être implantés. Il n'y a donc pas pour le moment d'analyse de terminaison.

D'autre part, les programmes Prolog qui sont pour le moment analysés ne doivent contenir ni prédicats de tests (var, list, ground) ni instructions arithmétiques. Ces opérations ne seront pas analysées. De même, le cut et la négation (!) ne sont pas encore considérés.

Le langage de spécification des comportements permet, quant à lui, de spécifier les comportements sous la forme de séquences abstraites. Parce que les composants E n'ont pas encore été implantés, il ne permet pas de définir les relations sur les termes et sur le nombre de solutions, que ce soit pour les substitutions abstraites ou les séquences abstraites.

D'un point de vue purement implantation, comme certaines fonctionnalités ont été reprise du système GAIA (qui était implanté en C), le code réutilisé a dans certains cas été traduit en C++ (sv, mo, frm, ps et toutes les opérations sur ces structures et sur les substitutions abstraites) mais d'autres sont restées implantées en C (lecture du programme prolog et normalisation de programmes). Toutes les nouvelles fonctionnalités sont quant à elles implantées en C++.

11. Trace complète de l'exécution de select/3

Nous présentons ici les résultats générés par l'analyseur[1] lorsqu'il considère la procédure select/3 :

```
liste(T).

select (X,L,LS) :- L = [H|T], H = X, LS=T, liste(T).
select (X,L,LS) :- L = [H|T], LS = [H|TS], select(X,T,TS).
```

et les comportements sont :

```
select/3 => in(X:var anylist, L:ground list, Ls:var anylist
                 :(X,X)(Ls,Ls):),
            ref(X:var anylist, [Y:ground any|Z:ground list]:ground list,
                 Ls:var anylist :(X,X)(Ls,Ls):),
            out(X:ground any, [Y:ground any|Z:ground list]:ground list,
                 LS:ground list : : )
            ,#Eout
            ,#Esol
            ,#size expr
liste/1  => in(L:ground list::), ref(L:ground list::), out(L:ground list::)
            ,#Eout ,#Esol ,#size expr
```

Les résultats retournés par l'analyseur sont :

[1] Tel qu'il est implanté pour le moment.

```
liste(X0) :-  [ 0 ]
.

select(X0 , X1 , X2) :-  [ 3 ]
     X3 =  [ X4 | X5 ] ,
     X1 = X3,
     X4 = X0,
     X2 = X5,
     liste( X5 ).
select(X0 , X1 , X2) :-  [ 5 ]
     X3 =  [ X4 | X5 ] ,
     X1 = X3,
     X6 =  [ X4 | X7 ] ,
     X2 = X6,
     select( X0 , X5 , X7 ).


================================================================
==================

Make sat...

================================================================
==================



================================================================
==================

Make sat done.
[
=====SAT OF select/3, <[
ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> ground,1 -> ground,2 -> ground,3 -> ground,4 ->
ground}
ty is :{0 -> any,1 -> list,2 -> list,3 -> any,4 -> list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====
 ]
>
,
=====SAT OF liste/1, <[
ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0}
mo is :{0 -> ground}
ty is :{0 -> list}
frm :{0 -> ?}
ps is :{}
psclose is :{}

Refined abstract substitution:

sv is :{0->0}
mo is :{0 -> ground}
ty is :{0 -> list}
```

```
frm :{0 -> ?}
ps is :{}
psclose is :{}

Output abstract substitution:

sv is :{0->0}
mo is :{0 -> ground}
ty is :{0 -> list}
frm :{0 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====
 ]
>
 ]
============================================================
==================

============================================================
==================

ANALYSE PROCEDURE...

============================================================
==================

AnalyseProcedure : input sequence is:

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}
```

```
Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> ground,1 -> ground,2 -> ground,3 -> ground,4 ->
ground}
ty is :{0 -> any,1 -> list,2 -> list,3 -> any,4 -> list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

============================================================
==================

============================================================
==================

ANALYSE CLAUSE...

============================================================
==================

ExtcSA.
[...]
AnalyseClause : B after EXTC(c, input(B)) :

ABSTRACT SEQUENCE :
Input abstract substitution:
```

```
sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2,3->3,4->4,5->5}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> var,4 -> var,5 ->
var}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> anylist,4 ->
anylist,5 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?,3 -> ?,4 -> ?,5 -> ?}
ps is :{(0,0)(2,2)(3,3)(4,4)(5,5)}
psclose is :{(0,0)(2,2)(3,3)(4,4)(5,5)}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

AnalyseClause : executing atom      X3 =  [ X4 | X5 ]
RESTRGSA.
[...]
AnalyseCLause : Substitution after RESTRG(1, B) :

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> var,2 -> var}
ty is :{0 -> anylist,1 -> anylist,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(1,1)(2,2)}
psclose is :{(0,0)(1,1)(2,2)}

AnalyseClause : Sequence after UNIF-func(brestr, inst) :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> var,2 -> var}
ty is :{0 -> anylist,1 -> anylist,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(1,1)(2,2)}
psclose is :{(0,0)(1,1)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> var,2 -> var}
ty is :{0 -> anylist,1 -> anylist,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(1,1)(2,2)}
psclose is :{(0,0)(1,1)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> ngv,1 -> var,2 -> var}
ty is :{0 -> anylist,1 -> anylist,2 -> anylist}
frm :{0 -> .(1,2),1 -> ?,2 -> ?}
ps is :{(1,1)(2,2)}
psclose is :{(1,1)(2,2)}

EOut component:

ESol component:

===== End of abstract sequence =====


ExtgSA.
[...]
```

```
AnalyseClause : Sequence after EXTG(1, B, B') :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2,3->3,4->4,5->5}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ngv,4 -> var,5 ->
var}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> anylist,4 ->
anylist,5 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?,3 -> .(4,5),4 -> ?,5 -> ?}
ps is :{(0,0)(2,2)(4,4)(5,5)}
psclose is :{(0,0)(2,2)(4,4)(5,5)}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

AnalyseClause : executing atom      X1 = X3
RESTRGSA.
[...]
```

```
AnalyseCLause : Substitution after RESTRG(1, B) :

sv is :{0->0,1->1}
mo is :{0 -> ground,1 -> ngv,2 -> var,3 -> var}
ty is :{0 -> list,1 -> anylist,2 -> anylist,3 -> anylist}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> ?}
ps is :{(2,2)(3,3)}
psclose is :{(2,2)(3,3)}

AiVarSA.
[...]
AnalyseClause : Sequence after UNIF-var(brestr, inst) :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1}
mo is :{0 -> ground,1 -> ngv,2 -> var,3 -> var}
ty is :{0 -> list,1 -> anylist,2 -> anylist,3 -> anylist}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> ?}
ps is :{(2,2)(3,3)}
psclose is :{(2,2)(3,3)}

Refined abstract substitution:

sv is :{0->0,1->1}
mo is :{0 -> ground,1 -> ngv,2 -> var,3 -> var,4 -> ground,5
-> ground}
ty is :{0 -> list,1 -> anylist,2 -> anylist,3 -> anylist,4 ->
any,5 -> list}
frm :{0 -> .(4,5),1 -> .(2,3),2 -> ?,3 -> ?,4 -> ?,5 -> ?}
ps is :{(2,2)(3,3)}
psclose is :{(2,2)(3,3)}

Output abstract substitution:

sv is :{0->0,1->0}
mo is :{0 -> ground,1 -> ground,2 -> ground}
ty is :{0 -> list,1 -> any,2 -> list}
frm :{0 -> .(1,2),1 -> ?,2 -> ?}
ps is :{}
psclose is :{}

EOut component:
```

```
ESol component:

===== End of abstract sequence =====

ExtgSA.
[...]
AnalyseClause : Sequence after EXTG(1, B, B') :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->4,3->1,4->2,5->3}
mo is :{0 -> var,1 -> ground,2 -> ground,3 -> ground,4 ->
var}
ty is :{0 -> anylist,1 -> list,2 -> any,3 -> list,4 ->
anylist}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(4,4)}
psclose is :{(0,0)(4,4)}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

AnalyseClause : executing atom      X4 = X0
RESTRGSA.
[...]
AnalyseCLause : Substitution after RESTRG(1, B) :

sv is :{0->0,1->1}
mo is :{0 -> ground,1 -> var}
ty is :{0 -> any,1 -> anylist}
frm :{0 -> ?,1 -> ?}
ps is :{(1,1)}
psclose is :{(1,1)}

AiVarSA.
[...]
AnalyseClause : Sequence after UNIF-var(brestr, inst) :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1}
mo is :{0 -> ground,1 -> var}
ty is :{0 -> any,1 -> anylist}
frm :{0 -> ?,1 -> ?}
ps is :{(1,1)}
psclose is :{(1,1)}

Refined abstract substitution:

sv is :{0->0,1->1}
mo is :{0 -> ground,1 -> var}
ty is :{0 -> any,1 -> anylist}
frm :{0 -> ?,1 -> ?}
ps is :{(1,1)}
psclose is :{(1,1)}

Output abstract substitution:

sv is :{0->0,1->0}
```

```
mo is :{0 -> ground}
ty is :{0 -> any}
frm :{0 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

ExtgSA.
[...]
AnalyseClause : Sequence after EXTG(1, B, B') :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->3,3->1,4->0,5->2}
mo is :{0 -> ground,1 -> ground,2 -> ground,3 -> var}
ty is :{0 -> any,1 -> list,2 -> list,3 -> anylist}
frm :{0 -> ?,1 -> .(0,2),2 -> ?,3 -> ?}
ps is :{(3,3)}
psclose is :{(3,3)}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

AnalyseClause : executing atom      X2 = X5
RESTRGSA.
[...]
AnalyseCLause : Substitution after RESTRG(1, B) :

sv is :{0->0,1->1}
mo is :{0 -> var,1 -> ground}
ty is :{0 -> anylist,1 -> list}
frm :{0 -> ?,1 -> ?}
ps is :{(0,0)}
psclose is :{(0,0)}

AiVarSA.
AiVarSubst.
Uact.
UactList.
Uact1.
Fcta.
ComputeSizeConst  :  A FAIRE
AnalyseClause : Sequence after UNIF-var(brestr, inst) :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1}
mo is :{0 -> var,1 -> ground}
ty is :{0 -> anylist,1 -> list}
frm :{0 -> ?,1 -> ?}
ps is :{(0,0)}
psclose is :{(0,0)}

Refined abstract substitution:
```

```
sv is :{0->0,1->1}
mo is :{0 -> var,1 -> ground}
ty is :{0 -> anylist,1 -> list}
frm :{0 -> ?,1 -> ?}
ps is :{(0,0)}
psclose is :{(0,0)}

Output abstract substitution:

sv is :{0->0,1->0}
mo is :{0 -> ground}
ty is :{0 -> list}
frm :{0 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

ExtgSA.
[...]
AnalyseClause : Sequence after EXTG(1, B, B') :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2,3->1,4->0,5->2}
mo is :{0 -> ground,1 -> ground,2 -> ground}
ty is :{0 -> any,1 -> list,2 -> list}
frm :{0 -> ?,1 -> .(0,2),2 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

AnalyseClause : executing atom      liste( X5 )
RESTRGSA.
[...]
AnalyseCLause : Substitution after RESTRG(1, B) :

sv is :{0->0}
mo is :{0 -> ground}
ty is :{0 -> list}
frm :{0 -> ?}
ps is :{}
psclose is :{}

LookUpSa.
[...]
AnalyseClause : Goal resolution:
Sequence in satp:

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0}
mo is :{0 -> ground}
ty is :{0 -> list}
```

```
frm :{0 -> ?}
ps is :{}
psclose is :{}

Refined abstract substitution:

sv is :{0->0}
mo is :{0 -> ground}
ty is :{0 -> list}
frm :{0 -> ?}
ps is :{}
psclose is :{}

Output abstract substitution:

sv is :{0->0}
mo is :{0 -> ground}
ty is :{0 -> list}
frm :{0 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

ExtgSA.
[...]
AnalyseClause : Sequence after EXTG(1, B, B') :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2,3->1,4->0,5->2}
mo is :{0 -> ground,1 -> ground,2 -> ground}
ty is :{0 -> any,1 -> list,2 -> list}
frm :{0 -> ?,1 -> .(0,2),2 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

RestrcSA.
[...]
AnalyseClause done.
Sequence Bout (after RESTRC(B, c):

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
```

```
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> ground,1 -> ground,2 -> ground}
ty is :{0 -> any,1 -> list,2 -> list}
frm :{0 -> ?,1 -> .(0,2),2 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

================================================================
==================

AnalyseProcedure : Concatenated sequence is :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
```

```
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> ground,1 -> ground,2 -> ground}
ty is :{0 -> any,1 -> list,2 -> list}
frm :{0 -> ?,1 -> .(0,2),2 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

================================================================
==================

ANALYSE CLAUSE...

================================================================
==================

ExtcSA.
[...]
AnalyseClause : B after EXTC(c, input(B)) :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
```

```
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2,3->3,4->4,5->5,6->6,7->7}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> var,4 -> var,5 ->
var,6 -> var,7 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> anylist,4 ->
anylist,5 -> anylist,6 -> anylist,7 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?,3 -> ?,4 -> ?,5 -> ?,6 -> ?,7 ->
?}
ps is :{(0,0)(2,2)(3,3)(4,4)(5,5)(6,6)(7,7)}
psclose is :{(0,0)(2,2)(3,3)(4,4)(5,5)(6,6)(7,7)}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

AnalyseClause : executing atom      X3 =  [ X4 | X5 ]
RESTRGSA.
[...]
AnalyseCLause : Substitution after RESTRG(1, B) :

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> var,2 -> var}
ty is :{0 -> anylist,1 -> anylist,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(1,1)(2,2)}
psclose is :{(0,0)(1,1)(2,2)}

AiFuncSa.
[...]
AnalyseClause : Sequence after UNIF-func(brestr, inst) :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> var,2 -> var}
ty is :{0 -> anylist,1 -> anylist,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(1,1)(2,2)}
psclose is :{(0,0)(1,1)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> var,2 -> var}
ty is :{0 -> anylist,1 -> anylist,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(1,1)(2,2)}
psclose is :{(0,0)(1,1)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> ngv,1 -> var,2 -> var}
ty is :{0 -> anylist,1 -> anylist,2 -> anylist}
frm :{0 -> .(1,2),1 -> ?,2 -> ?}
ps is :{(1,1)(2,2)}
psclose is :{(1,1)(2,2)}

EOut component:

ESol component:

===== End of abstract sequence =====

ExtgSA.
[...]
AnalyseClause : Sequence after EXTG(1, B, B') :

ABSTRACT SEQUENCE :
Input abstract substitution:
```

```
sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2,3->3,4->4,5->5,6->6,7->7}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ngv,4 -> var,5 ->
var,6 -> var,7 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> anylist,4 ->
anylist,5 -> anylist,6 -> anylist,7 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?,3 -> .(4,5),4 -> ?,5 -> ?,6 -> ?,7
-> ?}
ps is :{(0,0)(2,2)(4,4)(5,5)(6,6)(7,7)}
psclose is :{(0,0)(2,2)(4,4)(5,5)(6,6)(7,7)}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

AnalyseClause : executing atom      X1 = X3
RESTRGSA.
[...]
AnalyseCLause : Substitution after RESTRG(1, B) :

sv is :{0->0,1->1}
mo is :{0 -> ground,1 -> ngv,2 -> var,3 -> var}
ty is :{0 -> list,1 -> anylist,2 -> anylist,3 -> anylist}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> ?}
ps is :{(2,2)(3,3)}
psclose is :{(2,2)(3,3)}

AiVarSA.
[...]
AnalyseClause : Sequence after UNIF-var(brestr, inst) :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1}
mo is :{0 -> ground,1 -> ngv,2 -> var,3 -> var}
ty is :{0 -> list,1 -> anylist,2 -> anylist,3 -> anylist}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> ?}
ps is :{(2,2)(3,3)}
psclose is :{(2,2)(3,3)}

Refined abstract substitution:

sv is :{0->0,1->1}
mo is :{0 -> ground,1 -> ngv,2 -> var,3 -> var,4 -> ground,5
-> ground}
ty is :{0 -> list,1 -> anylist,2 -> anylist,3 -> anylist,4 ->
any,5 -> list}
frm :{0 -> .(4,5),1 -> .(2,3),2 -> ?,3 -> ?,4 -> ?,5 -> ?}
ps is :{(2,2)(3,3)}
psclose is :{(2,2)(3,3)}

Output abstract substitution:

sv is :{0->0,1->0}
mo is :{0 -> ground,1 -> ground,2 -> ground}
ty is :{0 -> list,1 -> any,2 -> list}
frm :{0 -> .(1,2),1 -> ?,2 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:
```

```
===== End of abstract sequence =====

ExtgSA.
[...]
AnalyseClause : Sequence after EXTG(l, B, B') :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->4,3->1,4->2,5->3,6->5,7->6}
mo is :{0 -> var,1 -> ground,2 -> ground,3 -> ground,4 ->
var,5 -> var,6 -> var}
ty is :{0 -> anylist,1 -> list,2 -> any,3 -> list,4 ->
anylist,5 -> anylist,6 -> anylist}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> ?,4 -> ?,5 -> ?,6 -> ?}
ps is :{(0,0)(4,4)(5,5)(6,6)}
psclose is :{(0,0)(4,4)(5,5)(6,6)}

EOut component:

ESol component:

===== End of abstract sequence =====
```

```
==========================================================
==================

AnalyseClause : executing atom      X6 =  [ X4 | X7 ]
RESTRGSA.
[...]
AnalyseCLause : Substitution after RESTRG(l, B) :

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> any,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

AiFuncSa.
[...]
AnalyseClause : Sequence after UNIF-func(brestr, inst) :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> any,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> any,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> ngv,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> any,2 -> anylist}
frm :{0 -> .(1,2),1 -> ?,2 -> ?}
```

```
ps is :{(2,2)}
psclose is :{(2,2)}

EOut component:

ESol component:

===== End of abstract sequence =====

ExtgSA.
[...]
AnalyseClause : Sequence after EXTG(1, B, B') :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->4,3->1,4->2,5->3,6->5,7->6}
mo is :{0 -> var,1 -> ground,2 -> ground,3 -> ground,4 ->
var,5 -> ngv,6 -> var}
ty is :{0 -> anylist,1 -> list,2 -> any,3 -> list,4 ->
anylist,5 -> anylist,6 -> anylist}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> ?,4 -> ?,5 -> .(2,6),6 -
> ?}
ps is :{(0,0)(4,4)(6,6)}
psclose is :{(0,0)(4,4)(6,6)}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

AnalyseClause : executing atom      X2 = X6
RESTRGSA.
[...]
AnalyseCLause : Substitution after RESTRG(1, B) :

sv is :{0->0,1->1}
mo is :{0 -> var,1 -> ngv,2 -> ground,3 -> var}
ty is :{0 -> anylist,1 -> anylist,2 -> any,3 -> anylist}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> ?}
ps is :{(0,0)(3,3)}
psclose is :{(0,0)(3,3)}

AiVarSA.
[...]
AnalyseClause : Sequence after UNIF-var(brestr, inst) :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1}
mo is :{0 -> var,1 -> ngv,2 -> ground,3 -> var}
ty is :{0 -> anylist,1 -> anylist,2 -> any,3 -> anylist}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> ?}
ps is :{(0,0)(3,3)}
psclose is :{(0,0)(3,3)}

Refined abstract substitution:

sv is :{0->0,1->1}
mo is :{0 -> var,1 -> ngv,2 -> ground,3 -> var}
ty is :{0 -> anylist,1 -> anylist,2 -> any,3 -> anylist}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> ?}
ps is :{(0,0)(3,3)}
```

```
psclose is :{(0,0)(3,3)}

Output abstract substitution:

sv is :{0->0,1->0}
mo is :{0 -> ngv,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> any,2 -> anylist}
frm :{0 -> .(1,2),1 -> ?,2 -> ?}
ps is :{(2,2)}
psclose is :{(2,2)}

EOut component:

ESol component:

===== End of abstract sequence =====

ExtgSA.
[...]
AnalyseClause : Sequence after EXTG(1, B, B') :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->4,3->1,4->2,5->3,6->4,7->5}
mo is :{0 -> var,1 -> ground,2 -> ground,3 -> ground,4 ->
ngv,5 -> var}
ty is :{0 -> anylist,1 -> list,2 -> any,3 -> list,4 ->
anylist,5 -> anylist}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> ?,4 -> .(2,5),5 -> ?}
ps is :{(0,0)(5,5)}
psclose is :{(0,0)(5,5)}

EOut component:

ESol component:

===== End of abstract sequence =====

============================================================
==================

AnalyseClause : executing atom      select( X0 , X5 , X7 )
RESTRGSA.
[...]
AnalyseCLause : Substitution after RESTRG(1, B) :

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

LookUpSa.
[...]
AnalyseClause : Goal resolution:
Sequence in satp:

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}
```

```
Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> ground,1 -> ground,2 -> ground,3 -> ground,4 ->
ground}
ty is :{0 -> any,1 -> list,2 -> list,3 -> any,4 -> list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

ExtgSA.
[...]
AnalyseClause : Sequence after EXTG(l, B, B') :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground,5 -> ground,6 -> ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list,5 -> any,6 -> list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> .(5,6),5 -> ?,6 -
> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->6,3->1,4->2,5->3,6->6,7->7}
mo is :{0 -> ground,1 -> ground,2 -> ground,3 -> ground,4 ->
ground,5 -> ground,6 -> ground,7 -> ground}
ty is :{0 -> any,1 -> list,2 -> any,3 -> list,4 -> any,5 ->
list,6 -> list,7 -> list}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> .(4,5),4 -> ?,5 -> ?,6 -
> .(2,7),7 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

RestrcSA.
[...]
AnalyseClause done.
Sequence Bout (after RESTRC(B, c):

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:
```

```
sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground,5 -> ground,6 -> ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list,5 -> any,6 -> list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> .(5,6),5 -> ?,6 -
> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->6}
mo is :{0 -> ground,1 -> ground,2 -> ground,3 -> ground,4 ->
ground,5 -> ground,6 -> ground,7 -> ground}
ty is :{0 -> any,1 -> list,2 -> any,3 -> list,4 -> any,5 ->
list,6 -> list,7 -> list}
frm :{0 -> ?,1 -> .(2,3),2 -> ?,3 -> .(4,5),4 -> ?,5 -> ?,6 -
> .(2,7)}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

ConsSeqAbs.
[...]
================================================================
==================

AnalyseProcedure : Concatenated sequence is :

ABSTRACT SEQUENCE :
Input abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var}
ty is :{0 -> anylist,1 -> list,2 -> anylist}
frm :{0 -> ?,1 -> ?,2 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Refined abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> var,1 -> ground,2 -> var,3 -> ground,4 ->
ground}
ty is :{0 -> anylist,1 -> list,2 -> anylist,3 -> any,4 ->
list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{(0,0)(2,2)}
psclose is :{(0,0)(2,2)}

Output abstract substitution:

sv is :{0->0,1->1,2->2}
mo is :{0 -> ground,1 -> ground,2 -> ground,3 -> ground,4 ->
var}
ty is :{0 -> any,1 -> list,2 -> list,3 -> any,4 -> list}
frm :{0 -> ?,1 -> .(3,4),2 -> ?,3 -> ?,4 -> ?}
ps is :{}
psclose is :{}

EOut component:

ESol component:

===== End of abstract sequence =====

================================================================
==================

================================================================
==================

AnalyseProcedure done.
================================================================
==================
```

12. Conclusion : vers un analyseur complet

Nous voici au terme de ce mémoire. Nous y avons décrit une méthode de vérification de programmes Prolog basée sur l'interprétation abstraite, qui a la particularité de s'inscrire dans une méthodologie de développement, et qui vise une utilisation plus sûre des programmes Prolog.

Nous avons en effet vu que les difficultés principales rencontrées lors de la mise au point et de l'utilisation de programmes Prolog sont la prédiction des comportements des procédures, et les corrections apportées à celles-ci afin d'obtenir des programmes corrects. Nous avons également vu que ces difficultés sont dues à la distance qui sépare la sémantique procédurale de la sémantique déclarative, et que cette distance est inévitable si l'on veut assurer une certaine efficacité aux programmes Prolog.

Nous avons ensuite étudié la méthodologie de développement proposée par Y. Deville, et par laquelle il est possible de réduire cette distance. En effet, on y propose de définir de manière formelle un ensemble de conditions d'application, et cela dès la spécification, et de dériver de manière systématique une version exécutable à partir d'une description logique indépendante de tout langage logique. Cette méthodologie vise la correction des programmes en proposant des outils de dérivation et de construction de programmes. Mais elle ne propose qu'une correction partielle de ceux-ci. En effet, la vérification des directionnalités est toujours à charge du programmeur. Et c'est ce type de vérification que prend en charge l'analyseur.

Nous donnons en effet la possibilité au programmeur de définir un ensemble de propriétés supplémentaires qui serviront de base à une analyse globale du programme. Ces propriétés consistent en la spécification des modes, des types, des formes, des partages de variables et de valeurs entre les termes, et des relations sur les tailles des termes présents dans une même substitution. Il est également possible de spécifier un ensemble de relations existantes entre les paramètres et les résultats d'une

procédure, ainsi que les propriétés liant le nombre de solutions d'une procédure aux paramètres utilisés pour exécuter cette procédure.

Le programmeur a ainsi la possibilité de définir un ensemble de comportements pour chaque procédure. Ces informations seront dès lors affinées et utilisées de manière intensive par l'analyseur. Celui-ci va alors interpréter le programme, en dégager les propriétés et comparer celles-ci aux spécifications données par l'utilisateur. Cette comparaison aura d'autant plus de valeur que le système met en corrélation les résultats capturés afin de les affiner mutuellement.

On le voit, une importante place est donnée aux connaissances qu'a l'utilisateur sur le programme. L'analyse va en effet considérer que ces connaissances sont des valeurs sûres et qu'elles doivent être vérifiées par le programme. Il s'agit donc d'un système orienté vers la vérification du programme. Cependant, ce système n'est pas fermé. En effet, les résultats de l'analyse sont tels qu'ils peuvent aisément servir de base à d'autres vérifications et optimisations[1]. On peut par exemple revoir comment il serait possible de dériver des implantations Prolog correctes rien qu'à partir de descriptions déclaratives. On peut aussi effectuer des analyses automatiques de complexité, et rechercher sur base de ces analyses quelle est la version la plus efficace d'une procédure Prolog.

Cependant, les capacités de ce système sont encore incomplètes si l'on vise une correction totale des programmes Prolog. En effet, l'analyseur tel qu'il est décrit actuellement ne prend pas en charge les instructions arithmétiques, les instructions cut, la négation par échec et les prédicats de test. Les built-ins arithmétiques et les prédicats de test peuvent facilement être intégrés sans ajout de code : il faut fournir les comportements qui capturent leur sémantique opérationnelle. Mais il peut être nécessaire de développer plus avant le domaine d'analyse et les opérations qui permettront de capturer les propriétés conséquentes aux instructions cut et négation par échec.

De même, l'analyseur ne capture de l'information que sur les suites finies de substitutions retournées par l'exécution d'un appel. Or, il y a de nombreuses classes de programme qui ne se terminent pas. Il faudra alors tenir compte des développements sur le domaine abstrait afin de considérer ces nombreux cas.

Il est également possible de développer une analyse d'occur-check sur base des domaines classiques (comme les partages de variables et les informations de linéarité), ce qui évite de considérer ce test comme une opération devant trouver une correspondance directe dans les opérations abstraites.

On le voit, l'analyse de programmes logiques est un domaine de recherche où beaucoup reste à faire. Les multiples analyses de correction qui peuvent se développer sont d'un intérêt indéniable quand on considère les projets de développement plus que conséquents que l'on rencontre à l'heure actuelle. D'autre part, les nombreuses possibilités d'optimisation pourraient enfin mener à une utilisation intensive de langages « intelligents » sans que cela ne se transforme inévitablement en une perte d'efficacité désolante.

---

[1] Qui, on le rappelle, sont nombreuses dans le cas des langages déclaratifs.

13. Références

[1] Baudouin Le Charlier, « Computational Logic », cours de 2eme Licence et Maîtrise en Informatique, Institut d'Informatique, FUDNP.

[2] Jean-Marie Jacquet, « Programmation Logique », cours de 1^{ere} Licence et Maîtrise en Informatique, Institut d'informatique, FUNDP.

[3] Richard A. O'Keefe, « The craft of Prolog », The MIT Press, 1990.

[4] Francis Giannesini, Henry Kanoui, Robert Pasero, Michel Van Caneghem, « Prolog », International Computer Science Series, 1986.

[5] Jean-Marie Jacquet (ed.) « Constructing logic programs », Jhon Willey & sons, 1993.

[6] L. Sterling, E. Shapiro, « The art of Prolog », Advanced Programming Techniques, The MIT Press, 1986.

[7] Yves Deville, « Logic programming : systematic program development », International Series in Logic Programming, Addison-Wesley, 1990.

[8] J. W. Lloyd, « Foundations of Logic Programming », Springer-Verlag, 2° Edition, 1987.

[9] Baudouin Le Charlier, Christophe Leclère, Sabina Rossi, Agostino Cortesi, « Automated verification of Prolog programs », The journal of Logic programing.

[10] Doran K. Wilde, « A library for doing polyhedral operations », Institut de recherche en informatique et systèmes aléatoires (IRISA), publication interne n°785, décembre 1993

Annexe A : Programmes Prolog normalisés

Un programme Prolog peut très vite avoir une structure très complexe. En effet, les têtes de clauses peuvent contenir des variables, atomes simples et atomes composés, le corps de la clause peut être composé d'atome foncteur composé d'un nombre quelconque d'atomes composants, eux-mêmes composés, etc. Cependant, il est toujours possible de traduire un programme logique en un programme logique normalisé.

Un programme normalisé est construit sur un ensemble de « variables de programme » $\{X_1,...,X_l,...\}$. Il est composé d'un ensemble de clauses de la forme $p(X_{i_1},...,X_{i_m}) :-l_1,...,l_p$. La tête de la clause est l'atome $p(X_{i_1},...,X_{i_m})$ », les X_{i_j} étant des variables de programmes. Le corps est « $l_1,...l_p$ ». Les littéraux l_i ont trois formes possibles :

i) $l_i \equiv q(X_{i_1},...X_{i_n})$ où $X_{i_1},...X_{i_n}$ sont des variables de programme distinctes ;

ii) $l_i \equiv X_{i_1} = X_{i_2}$ où X_{i_1} et X_{i_2} sont des variables de programme distinctes ;

iii) $l_i \equiv X_{i_1} = f(X_{i_2},...X_{i_n})$ où f est une fonction d'arité n-1 et $X_{i_2},...X_{i_n}$ sont des variables de programme distinctes.

Grâce aux programmes normalisés, on ne manipule que deux cas simples d'unification : l'unification de deux variables et l'unification d'une variable à une fonction. De plus, les prédicats et les fonctions ont toujours des variables de programme pour paramètres. Ces paramètres ne sont jamais des termes aux formes compliquées. Enfin, les substitutions en entrée et en sortie pour une procédure p/n sont toujours exprimées en termes de variables de programme $X_1,...X_n$, ce qui simplifie fortement les problèmes de renomination de variables.

Par exemple, la procédure select/3 étant :

select(H, [H|T], T) :- list(T).
select(X, [H|T], [H|TS]) :- select(X,T,TS).

La forme normalisée de select/3 est :

select(X_1,X_2,X_3) :-X_2=[X_1|X_3], list(X_3).
select(X_1,X_2,X_3) :-X_2=[X_4|X_5], X_3=[X_4|X_6], select(X_1,X_5,X_6).

Annexe B : Langage de description des comportements

Le langage décrit ici va permettre à l'utilisateur de spécifier l'ensemble des comportements qu'il veut associer aux procédures. Cette spécification se fera dans un fichier séparé du code Prolog.

<behaviours> : := | <behaviour> <behaviours>
<behaviour> : := <procedure name> / <arity> <liste abstr. seq.>
<procedure name> : := <minuscule><caracteres>
<liste abstr. seq.> : := | => <input abstr. subst.>, <ref abstr. subst>, <out abstr. subst.>, <Eout>, <Esol>, <size expr> <liste abstr. seq.>
<input abstr. subst.> : := in (<abstr. subst.>)
<ref abstr. subst.> : := ref (<abstr. subst.>)
<out abstr. subst.> : := out (<abstr. subst.>)
<abstr. subst.> : := <liste arguments> : <sharing decl.> : <size relation>
<liste arguments> : := | <argument> | <argument><liste arguments>
<argument>[1] : := <variable decl.> | <list decl.> | <functor decl.> | _
<variable> : := <majuscule><caracteres> | _<caracteres>
<variable decl.> : := <variable> : <mode> : <type>
<functor decl.> : := <functor> (<list of arguments>) : <mode> <type>
<list decl.> : := [<liste arguments>] : <mode> <type>
| [<liste arguments> | <argument>] : <mode> <type>
<identifier> : := <caracteres>
<mode> : := bottom | ground | var | ngv | gv | nv | ng | any
<type> : := bottom | list | anylist | any
<functor> : := <minuscule><caracteres> | <integer><caracteres>
<sharing decl.> : := | (<variable> , <variable>) <sharing decl.>
<size relation> : := | bottom | <inequality> , <size relation>
<inequality> : := <tag expression> = <tag expression>
| <tag expression> = <tag expression>
<size expr.> : := <expression>

[1] On ne peut jamais avoir un symbole de prédicat dans la liste des arguments d'une substitution abstraite.

<Eout> : := <size relation>
<Esol> : := <size relation>
<arity> : := <integer>
<expression> : := <identifier> | <integer> | <expression> <operator> <expression>
<operator> : := + | - | *
<tag> : := _in | _ref | _out
<tag identifier> : := <identifier><tag>
<tag expression> : := <tag identifier> | <integer> | sol | <tag expression> <operator> <tag expression>
<caracteres> : := | <caractere><caracteres>
<caractere> : := <minuscule> | <majuscule> | <integer> | <caract. speciaux>
<minuscule> : := a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<majuscule> : := A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<entier> : := <chiffre> | <chiffre><entier>
<chiffre> : := 0|1|2|3|4|5|6|7|8|9
<caract. speciaux> : := : | [|] | (|) | - | * | / | + | _ | = | < | > | #

Il est également possible d'insérer des commentaires dans les fichiers de spécification des comportements. Si on insère le caractère # à n'importe quelle position dans une ligne du fichier, les caractères suivants sur la même ligne sont des commentaires.

Annexe C : Notions propres à l'implantation

NOTIONS DE BASE

On note :

- P un ensemble fini de symboles de prédicats ;
- T l'ensemble de tous les termes ;
- $I = \{1,...,n\}$ un ensemble d'index ($I \subset \aleph$) ;
- T_I est l'ensemble de tous les tuples de termes $\langle t1,...,tn \rangle = \langle t_i \rangle_{i \in I}$;
- T_I^* l'ensemble de tous les termes de la forme $f(i_1,...,i_n)$ où f est une fonction d'arité n, $i_1,...,i_n \in I$.
- une size measure est une fonction $\|.\| : T \rightarrow \aleph$;
- Subst(S) est l'ensemble de toutes les substitutions contenues dans la séquence de substitutions S.

Opérateur de composition ▫

Soient f_1 et f_2 deux fonctions de domaine D_1 et D_2, et telles que : $\forall d \in D_1 \cap D_2 : f_1(d)=f_2(d)$. On note $f_1 \text{ ▫ } f_2$ la fonction f de domaine $D_1 \cup D_2$ et telle quel $\forall d \in D_i : f(d) = f_i(d)$ (i = 1,2).

Exemple :

f1 :		*f2 :*		*f= f1▫f2 :*	
1 →	*a*	*1* →	*a*	*1* →	*a*
2 →	*b*	*2* →	*b*	*2* →	*b*
3 →	*c*	*4* →	*d*	*3* →	*c*
		5 →	*e*	*4* →	*d*
				5 →	*e*

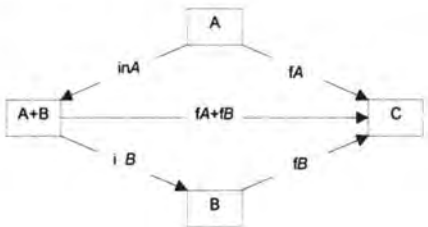
Union disjointe, somme disjointe

Soient A et B deux ensembles (non disjoints). L'**union disjointe** de A et B est un ensemble, noté A+B, doté de deux fonctions d'injection in_A et in_B tels que :

$\forall$ ensemble C : $\forall$ f_A :A→C, f_B : B→C : $\exists$ 1! f : A+B→C : f_A = f o in_A, f_B = f o in_B.

Puisque f est définie de manière unique, on la dénote par f_A et f_B. On notera f_A+f_B la **somme disjointe** de f_A et f_B.

On a la commutativité suivante :



Exemple :

A = {1,2,3,4}, B = {1,3,5,7}, C = {a,b,c,d,e}

inA :	*A→*	*A+B*
	1→	*1*
	2→	*2*
	3→	*3*
	4→	*4*

inB :	*B→*	*A+B*
	1→	*1*
	3→	*3*
	5→	*5*
	7→	*6*

fA :	*A→*	*C*
	1→	*a*
	2→	*b*
	3→	*c*
	4→	*d*

fB :	*B→*	*C*
	1→	*a*
	3→	*c*
	5→	*d*
	7→	*e*

f:	*A+B→*	*C*
	1→	*a*
	2→	*b*
	3→	*c*
	4→	*d*
	5→	*d*
	6→	*e*

Expressions linéaires

Si V est un ensemble de variables, on note EXP_V l'ensemble de toutes les expressions linéaires avec des coéfficients entiers sur les variables de V.

Soient V' un autre ensemble de variables, l'expression $exp(X_1,...,X_n) \in EXP_V$ ($X_1,...,X_n \in V$) et une fonction f de V dans V', on dénote paf $\hat{f}$ la fonction de EXP_V dans $EXP_{V'}$ définie par $\hat{f}(exp(X_1,...,X_n))=exp(f(X_1),...,f(X_n))$.

Exemple :

V = {X,Y,Z}, V' = {X',Y',Z',W'}, exp_V = X+2Y+Z

f: *V→* *V'* *$\hat{f}$:* *Exp_V→* *$Exp_{V'}$*

$X \rightarrow Y'$	$\ldots \rightarrow \ldots$
$Y \rightarrow Z'$	$X+2Y+Z \rightarrow Y'+2Z'+W'$
$Z \rightarrow W'$	

STRUCTURAL MAPPINGS

Les structurals mappings sont utilisés pour formaliser les correspondances entre les indices de différents composants dans la même séquence abstraite. Un structural mapping entre deux pseudo substitutions abstraites est un mapping entre les indices correspondants préservant les informations contenues dans sv et frm. On appelle ce mapping « structurel » parce qu'il respecte la structure donnée par les composant frm.

Soient :
β = (sv,mo,ty,frm,ps,E) une substitution abstraite définie sur I ;
β' = (sv',mo',ty',frm,'ps',E') une substitution abstraite définie sur I'.

Un structural mapping entre β et β', *s'il existe*, est une fonction tr : I→I' telle que :

$\forall X \in dom(\beta)$, $tr(sv(X)) = sv'(X)$;
$\forall i \in I : frm(i) = f(i_1,\ldots,i_n) \Rightarrow frm'(tr(i)) = f(tr(i_1),\ldots tr(i_n))$.

Exemple : tr est le structural mapping de β et β' :

β : sv	frm	β' : sv'	frm'	tr : I→	I'
X1→1	1→[2|3]	X1→4	1→?	1	4
X2→4	2→?	X2→7	2→?	2	2
X3→7	3→[]	X3→8	3→[]	3	3
	4→?		4→[2|3]	4	7
	7→[2|8]		5→[]	7	8
	8→?		6→?	8	9
	9→?		7→?	9	*undef*
			8→[2|9]		
			9→[2|5]		

CONSTRAINED MAPPING

Les constrained mappings permettent de manipuler des indices. Leur rôle est important car ils définissent la correspondance entre des termes équivalents dans des composants différents. Par exemple, les termes définis dans une substitution ont tous un index. Mais si on considère deux substitutions, les index y évoluent de manière indépendante. Il ne suffit pas d'avoir l'index d'un terme dans une substitution pour que

cet index identifie le terme équivalent dans l'autre substitution. Il faut posséder en même temps les index d'un même terme dans les deux substitutions pour établir une correspondance. C'est ce que permet de réaliser le constrained mapping. On définit une fonction qui va d'un ensemble d'index dans un autre et qui établit la correspondance entre les index des deux ensembles. Cette fonction peut être « parcourue dans les deux sens » afin de retrouver l'index correspondant à partir de n'importe quel ensemble de départ. De plus, on définira « l'application d'un constrained mapping sur un composant » l'opération qui consiste à exprimer un composant défini sur un ensemble d'index en un composant équivalent sur un autre ensemble d'index. Cependant, cette opération peut provoquer certaines pertes de précision. C'est dû au fait que la fonction tr peut ne pas définir d'indice équivalent pour un indice donné. Des exemples illustrant ce cas seront donnés par la suite.

Soient :
I et I' deux ensembles finis d'index ;
tr : I→I' une fonction.

Le constrained mapping de tr au niveau concret est la paire de fonctions duales :
$tr^{>}_{*} : \wp(T^{I}) \rightarrow \wp(T^{I'})$ et
$tr^{<}_{*} : \wp(T^{I'}) \rightarrow \wp(T^{I})$ telles que, $\forall \Sigma_I \in \wp(T^{I}), \forall \Sigma_{I'} \in \wp(T^{I'})$:

$tr^{>}_{*}(\Sigma^{I}) = \{\langle s_i \rangle_{i \in I'} \in T^{I} \mid \exists \langle t_i \rangle_{i \in I} \in \Sigma^{I} : \forall i \in I, s_{tr(i)} = t_i\}$

$tr^{<}_{*}(\Sigma^{I'}) = \{\langle t_i \rangle_{i \in I} \in T^{i} \mid \exists \langle s_i \rangle_{i \in I'} \in \Sigma^{I'} : \forall i \in I, t_i = s_{tr(i)}\}$

La fonction $tr^{>}_{*}$, lorsqu'elle est appliquée sur un ensemble de tuples de termes $\langle t_1,...,t_n \rangle$ définis sur I, retourne un ensemble de tuple de termes $\langle s_1,...,s_m \rangle$ définis sur I' équivalents par rapport à leur indice. La fonction $tr^{<}_{*}$, lorsqu'elle est appliquée sur un ensemble de tuples de termes $\langle s_1,...,s_m \rangle$ définis sur I, retourne un ensemble de tuple de termes $\langle t_1,...,t_n \rangle$ définis sur I' équivalents par rapport à leur indice.

Soient :
AI et AI' deux domaines abstraits approximant $\wp(T^{I})$ et $\wp(T^{I'})$;
une fonction de concrétisation Cc.

Le constrained mapping de tr au niveau abstrait abstrait est une paire d'approximations $tr^{>} : A_I \rightarrow A_{I'}$ et $tr^{<} : A_{I'} \rightarrow A_I$ d'un constrained mapping concret telles que :

$\forall \alpha_I \in A_I, tr^{>*}(Cc(\alpha_I)) \subseteq Cc(tr^{>}(\alpha_I))$

$\forall \alpha_{I'} \in A_{I'}, tr^{<*}(Cc(\alpha_{I'})) \subseteq Cc(tr^{<}(\alpha_{I'}))$.

Constrained mapping sur le composant mo

Soient :
tr : I'→I une fonction partielle entre deux ensembles d'indices ;
mo : I→Modes un élément de $Modes_I$;
mo' : I'→Modes un élément de $Modes_{I'}$.

$tr^{<}(mo) = mo'$.

$tr^{<}$: $Modes_I \rightarrow Modes_{I'}$
$(I \rightarrow Modes) \rightarrow (I' \rightarrow Modes)$: $\forall i \in I'$:
$mo'(i) =$ $mo(tr(i))$ si $tr(i)$ est défini
any sinon.

L'opération respecte la condition suivante :

Si $(t_i)_{i \in I} \in Cc(mo)$
$j \in I'$
$tr(j)$ est défini
$t_{tr(j)} = t'_i$
Alors $(t_j')_{j \in I'} \in Cc(mo') \wedge \forall\, mo'' : (t_j')_{j \in I'} \in Cc(mo'') : mo' \leq mo''$.

Exemple :

I = {1,2,3}, I' = {4,5,6,7}

mo :	*I →*	*Modes*
	1 →	*var*
	2 →	*ground*
	3 →	*gv*

tr :	*I' →*	*I*
	4 →	*1*
	5 →	*2*
	6 →	*2*

mo'	*I' →*	*Modes*
	4 →	*var*
	5 →	*ground*
	6 →	*ground*
	7 →	*any*

$\underline{tr^{>}(mo') = mo.}$

$tr^{>}$: $Modes_{I'} \rightarrow Modes_I$
$(I' \rightarrow Modes) \rightarrow (I \rightarrow Modes)$: $\forall i \in I$:

$mo(i) =$ $GLB_{mode}(mo'(j_1),\ldots,mo'(j_n))$
si $tr(j_1)=\ldots=tr(j_n) = i$ $(j_l \in I')$
any sinon $(\neg \exists\, j \in I' : tr(j)=i)$

Exemple :

I = {1,2,3}, I' = {4,5,6,7}

mo' :	*I' →*	*Modes*
	4 →	*var*
	5 →	*ground*
	6 →	*ground*
	7 →	*any*

tr :	*I' →*	*I*
	4 →	*1*
	5 →	*2*
	6 →	*2*

mo	*I →*	*Modes*
	1 →	*var*
	2 →	*ground*
	3 →	*any*

On a ici un exemple de perte de précision décrit dans l'introduction. On voit que la fonction tr de cet exemple est identique à la fonction tr de l'exemple précédent, et que le composant mo' utilisé ici est le résultat du premier exemple. Mais le mode du terme indicé par « 3 » est devenu « any » alors qu'il était « gv » au départ. Cette perte de précision est due à la fonction tr qui ne définit pas de correspondance pour un indice « 3 » appartenant à I.

Constrained mapping sur le composant ty

Soient :
tr : I'→I une fonction partielle entre deux ensembles d'indices ;
ty : I→Types un élément de $Types_I$;
ty' : I'→Types un élément de $Types_{I'}$.

$\underline{tr^{<}(ty) = ty'.}$

$tr^{<}$: $Types_I$→ $Types_{I'}$
(I→Types) → (I'→Types) : ∀ i ∈ I' :
ty'(i) = ty(tr(i)) si tr(i) est défini
any sinon.

L'opération respecte la condition suivante :

Si $(t_i)_{i \in I} \in Cc(ty)$
$j \in I'$
tr(j) est défini
$t_{tr(j)} = t'_i$
Alors $(t_j')_{j \in I'} \in Cc(ty') \wedge \forall\ ty'' : (t_j')_{j \in I'} \in Cc(ty'') : ty' \leq ty''$.

Exemple :

I = {1,2,3}, I' = {1,2,4,5}

ty :	*I→*	*Types*
	1→	*any*
	2→	*list*
	3→	*anylist*

tr :	*I'→*	*I*
	1→	*1*
	4→	*2*
	5→	*2*

ty'	*I'→*	*Types*
	1→	*any*
	2→	*any*
	4→	*list*
	5→	*list*

$\underline{tr^{>}(ty') = ty.}$

$tr^{>}$: $Types_{I'}$ → $Types_I$
(I'→Types) → (I→Types) : ∀ i ∈ I :

ty(i) = $GLB_{type}(ty'(j_1),\ldots,ty'(j_n))$
si $tr(j_1)=\ldots=tr(j_n) = i$ ($j_l \in I'$)
any sinon ($\neg\exists\ j \in I' : tr(j)=i$)

Exemple :

I = {1,2,3}, I' = {1,2,4,5}

ty'	*I'→*	*Types*
	1→	*any*
	2→	*any*
	4→	*list*
	5→	*list*

tr :	*I'→*	*I*
	1→	*1*
	4→	*2*
	5→	*2*

ty	*I→*	*Types*
	1→	*any*
	2→	*list*
	3→	*any*

Constrained mapping sur le composant ps

Soient :
tr : I'→I une fonction partielle entre deux ensembles d'indices ;
ps ⊆ IX I un élément de $PSharing_I$;
ps' ⊆ I' X I' un élément de $PSharing_{I'}$.

$tr^{<}(ps) = ps'$.

$tr^{<}$: $PSharing_I$ → $PSharing_{I'}$
ps ⊆ IXI → ps' ⊆ I'XI' : ∀ i,j ∈ I' :
(i,j) ∈ ps' ⇔ tr(i) et tr(j) sont définis
et (tr(i),tr(j)) ∈ ps.

L'opération respecte la condition suivante :

Si $(t_i)_{i \in I} \in Cc(ps)$
$j \in I'$
tr(j) est défini
$t_{tr(j)} = t'_i$
Alors $(t'_j)_{j \in I'} \in Cc(ps') \wedge \forall\ ps'' : (t'_j)_{j \in I'} \in Cc(ps'') : ps' \leq ps''$.

Exemple :

ps = {(1,1),(1,2),(2,3),(4,5)}, I = {1,2,3,4,5}, I' = {1,2,3}

tr : I'→ I
1→ 1
2→ 3
3→ 3

ps' = {(1,1)}

$tr^{>}(ps') = ps$.

$tr^{>}$: $PSharing_{I'}$ → $PSharing_I$
ps ⊆ IXI → ps' ⊆ I'XI' : ∀ i,j ∈ I' :
ps(tr(i),tr(j)) = ps'(i,j) ⇔ tr(i) et tr(j) sont définis
et ps'(i,j)

Exemple :

ps' = {(1,2),(3,1),(2,4)}, I = {1,2,3,4,5}, I' = {1,2,3,4}

tr : I'→ I
1→ 1
2→ 3
3→ 3

ps = {(1,3),(3,1)}

Annexe D : Implantation de l'algorihtme abstrait

CONSTRUCTION DU SAT

MakeSat(Sbeh) = sat.

Soient :
Sbeh un ensemble de behaviours de la forme $\langle p, \{\langle B_1,se_1\rangle,...,\langle B_q,se_q\rangle\}\rangle$ ($p \in P$).

L'implantation est :

```
for all ⟨p, {⟨B1,se1⟩,...,⟨Bq,seq⟩}⟩ ∈ Sbeh do
        satp = {B1,...,Bq}
        n = q ;
        m = q
        for i = 1 to n do
                for j = i+1 to m do
                        B = JOIN_SA(Bi,Bj)
                        if B ∉ satp then
                                n = n + 1
                                satp = satp ∪ {B}
        m = n.
```

EXTENSION D'UNE SÉQUENCE ABSTRAITE AUX VARIABLES D'UNE CLAUSE

EXTC_SA(c,β) = B'.

Soient :
c une clause ;
$D = \{X_1,...,X_n\}$ l'ensemble des variables de tête de c ;
$D' = \{X_1,...,X_m\}$ ($n \leq m$) l'ensemble des variables de c ;
$\beta = \langle sv, mo, ty, frm, ps, E \rangle$ définie sur I ;
$dom(\beta) = D$;
$B' = \langle \beta_{in}', \beta_{ref}', \beta_{out}', E_{ref\_out}' E_{sol}' \rangle$;

$dom_{in}(B') = D$;
$dom_{out}(B') = D'$.

L'implantation est :

$\beta_{in}' = \beta$
$\beta_{ref}' = \beta$
$\beta_{out}' = EXTC\_SUBST(c,\beta)$ $(\Rightarrow \exists\ tr : I \rightarrow I_{out}')$
$in_{ref} : I \rightarrow I + I_{out}'$
$in_{out} : I_{out}' \rightarrow I + I_{out}'$
$E_{ref\_out}' = \{(sz \circ in_{ref})(i) = (sz \circ in_{out} \circ tr)(i)\}$
$E_{sol}' = \{sol = 1\}$

EXTENSION D'UNE SUBSTITUTION ABSTRAITE AUX VARIABLES D'UNE CLAUSE

EXTC_SUBST(c,β) = β'.

Soient :
c une clause ;
$D = \{X_1,...,X_n\}$ l'ensemble des variables de tête de c ;
$D' = \{X_1,...,X_n,X_{n+1},...,X_{n+k}\}$ l'ensemble des variables de c ;
$\beta = \langle sv,mo,ty,frm,ps,E \rangle$ définie sur $I_p=\{1,...,p\}$;
$dom(\beta) = D$;
$\beta' = \langle sv',mo',ty',frm',ps',E' \rangle$ définie sur $I_{p'}=\{1,...,p'\}$;
$dom(\beta') = D'$;
$\{X_{n+1},...,X_{n+k}\} = D' / D$ = l'ensemble des variables à ajouter dans β.

L'implantation de l'opération est :

$\forall\ i : 1 \le i \le n$: $sv'(X_i) = sv(X)$
$\forall\ i : n+1 \le i \le n+k$: $sv'(X_i) = p+i$
$\forall\ i : 1 \le i \le p$: $mo'(i) = mo(i)$
$ty'(i) = ty(i)$
$\forall\ i : p+1 \le i \le p+k$: $mo'(i)=var$
$ty'(i) = anylist$
$frm' = frm$
$ps' = ps \cup \{(i,i) : i \in \{p+1,...,p+k\}\}$
E' = ? ? ?

RESTRICTION D'UNE SÉQUENCE ABSTRAITE AUX TERMES D'UN ATOME

RESTRG_SA(l,B) = β'.

Soient :
l un littéral de la forme $p(X_{i_1},...X_{i_n})$, ou de la forme $X_{i_1}=X_{i_2}$ (n=2) ou de la forme $X_{i_1}=f(X_{i_2},...,X_{i_n})$;
$B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$;
$dom_{out}(B) = \{X_1,...,X_m\}$;
$\{X_{i_1},...,X_{i_n}\} \subseteq \{X_1,...,X_m\}$; $(n \le m)$;
$\beta' = \langle sv', mo', ty', frm', ps', E' \rangle$.

L'implantation est :

$\beta' = RESTRG\_SUBST(l, \beta_{out})$.

RESTRICTION D'UNE SUBSTITUTION ABSTRAITE AUX TERMES D'UN ATOME

RESTRG_SUBST(l,β) = β'.

Soient :
β= ⟨ sv,mo,ty,frm,ps,E ⟩ définie sur I ;
dom(β) = $\{X_1,...,X_n\}$;
l un littéral de la forme $p(X_{i_1},...X_{i_n})$, ou de la forme $X_{i_1}=X_{i_2}$ (n=2) ou de la forme $X_{i_1}=f(X_{i_2},...,X_{i_n})$;
β'= ⟨ sv',mo',ty',frm',ps',E' ⟩ définie sur I' ;
dom(β') = $\{X_1,...,X_m\}$;
tr :I→I'.

L'opération respecte la condition :

$\forall \theta \in Cc(\beta) : \theta=\{X_1/t_1,...,X_n/t_n\} \Rightarrow \{X_1/t_{i_1},...,X_m/t_{i_m}\} \in Cc(\beta')$.

L'implantation procède en deux étapes :

i) projection de β sur $\{X_{i_1},...,X_{i_m}\}$. Cette projection donne la substitution abstraite β_{tmp} définie seulement sur les variables apparaissant dans l ;
ii) création d'un mapping tr des indices x_{i_k} sur les indices x_k. Ce mapping permet d'exprimer β_{tmp} en termes de $\{X_1,...X_m\}$, soit β'.

INTERPRÉTATION D'UN ATOME

Unification de variables

Unification de variables dans une substitution abstraite

AI_VAR_SUBST(β) = ⟨ β', ss, sf, tr, U ⟩ .

Soient :
β= ⟨ sv,mo,ty,frm,ps,E ⟩ définie sur I ;
β'= ⟨ sv',mo',ty',frm',ps',E' ⟩ définie sur I' ;
dom(β) = dom(β') = $\{X_1,X_2\}$;
ss et sf deux valeurs booléennes ;
tr : I→I' le structural mapping de β sur β' ;
$U \subseteq I$.

L'opération s'implante comme suit :

⟨ β',tr,ss,sf,U ⟩ = UNIF($sv(X_1)$, $sv(X_2)$, β)

Unification de variables dans une séquence abstraite

AI_VAR_SA(β) = B.

Soient :
$\beta = \langle sv,mo,ty,frm,ps,E \rangle$;
$dom(\beta) = \{X_1,X_2\}$;
$B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$ une séquence pseudo-abstraite.

L'implantation est :

$\beta_{in} = \beta$
tr_{in_ref} une inclusion canonique

On réutilise tout d'abord l'opération AI_VAR_SUBST pour calculer la substitution abstraite résultant de l'unification de X_1 et X_2 dans la substitution β. On établit aussi un mapping tr entre les indices de β et les indices correspondant de β_{out}. Ce mapping permettra de raffiner β à l'ensemble des substitutions concrètes $\theta \in Cc(\beta)$ dans lesquelles l'unification de Xi et Xj n'échoue jamais.

$\langle \beta_{out}, tr, ss, sf, U \rangle$ = AI_VAR_SUBST(β)

On se base sur tr pour raffiner les informations sur les modes, types et patterns contenues dans β pour construire β_{ref}. Ce raffinement se fait à l'aide de l'opération Ref_{Ref}. On dérive aussi les contraintes entre les tailles des termes de β_{ref} et β_{out} ainsi que les contraintes sur le nombre de solutions.

si (ss) alors
  $\beta_{ref} = \beta_{in}$
  $tr_{ref\_out} = tr$
  $E_{ref\_out} = \{[[sz(in_{ref}(i))=sz(in_{out}(tr_{ref\_out}(i)))]] : i \in tr_{in\_ref}(U)\}$
  $E_{sol} = \{ [[sol = 1]] \}$
si (sf) alors
  $\beta_{ref} = \bot$
  tr_{ref_out} = undef
  $E_{ref\_out} = \bot = \{ [[sol = 0]] \}$
  $E_{sol} = \bot$
si $\neg$(sf $\vee$ ss) alors
  $(\beta_{ref}, tr_{ref\_out}) = Ref_{Ref}(\beta_{in},\beta_{out},tr)$
  $E_{ref\_out} = \{[[sz(in_{ref}(i))=sz(in_{out}(tr_{ref\_out}(i)))]] : i \in tr_{in\_ref}(U)\}$
  $E_{sol} = \{[[0 \leq sol]], [[sol \leq 1]]\}$

Unification d'une variable et d'un foncteur

Unification d'une variable et d'un foncteur dans une substitution abstraite

AI_FUNC_SUBST(β,f) = $\langle \beta', ss, sf, tr, U \rangle$.

Soient :
$\beta = \langle sv,mo,ty,frm,ps,E \rangle$ définie sur $I_p=\{1,...,p\}$;
$dom(\beta) = \{X_1,...,X_n\}$
f est un symbole de fonction d'arité n-1 ;
$\beta' = \langle sv',mo',ty',frm',ps',E' \rangle$ définie sur $I_p=I_p \cup \{p+1\}$;

ss et sf sont des valeurs booléennes ;
tr est un mapping entre β et β' ;
U est l'ensemble des indices non affectés par l'unification.

L'implantation de cette opération se déroule en plusieurs étapes :

(1) On crée avant tout une subsitution intermédiaire β_{tmp} égale à β mais dans laquelle on ajoute un subterm indexé par p+1 et qui a la forme f(sv(X_2),...sv(X_n)).
On a : $\beta_{tmp} = \langle sv_{tmp}, mo_{tmp}, ty_{tmp}, frm_{tmp}, ps_{tmp}, E_{tmp} \rangle$ sur $I_{p,tmp}=\{1,\ldots,p+1\}$
$dom(\beta_{tmp}) = dom(\beta)$

$\forall i : 1 \le i \le n : sv_{tmp}(X_i) = sv(X_i)$
$\forall i : 1 \le i \le p : \quad mo_{tmp}(i) = mo(i)$
$ty_{tmp}(i) = ty(i)$
$mo_{tmp}(p+1) = CONS_{mode}(f, mo(sv(X_2)),\ldots,mo(sv(X_n))))$
$ty_{tmp}(p+1) = CONS_{type}(mo_{tmp}(p+1)f,ty(sv(X_2)),\ldots,ty(sv(X_n))))$
$frm_{tmp} = frm \cup \{\langle p+1, f(sv(X_2),\ldots,sv(X_n)) \rangle\}$
$ps_{tmp} = ps$
$E_{tmp} = E$

(2) Ensuite, on unifie X_1 à ce nouveau subterm p+1.

$\langle \beta', ss, sf, tr, U \rangle = UNIF(sv(X_1), p+1, \beta_{tmp})$

On voit que le mode et le type du terme p+1 sont les meilleures approximations possibles du terme composé à partir des termes le composant.

Unification d'une variable et d'un foncteur dans une séquence abstraite

AI_FUNC_SA(β,f) = B. Cette opération unifie X_1 et f(X_2,..X_n) dans β et retourne la séquence abstraite B contenant le résultat.

Soient :
$\beta = \langle sv, mo, ty, frm, ps, E \rangle$;
$dom(\beta)=\{X_1,\ldots,X_n\}$;
f est un symbole de fonction d'arité n-1 ;
$B = \langle \beta_{in}, \beta_{out}, \beta_{ref}, E_{ref\_out}, E_{sol} \rangle$.

L'implantation est :

$\langle \beta_{out}, ss, sf, tr, U \rangle$ = AI_FUNC_SUBST(β,f)
$\beta_{in} = \beta$
si (ss) alors
    $\beta_{ref} = \beta_{in}$
    E_{ref_out} = COMPUTE_SZ_CONST(β_{in},β_{out})
    $E_{sol} = \{[[sol = 1]]\}$
si (sf) alors
    $\beta_{ref} = \bot$
    $E_{ref\_out} = \varnothing$
    $E_{sol} = \{[[sol = 0]]\}$
si $\neg$(sf $\vee$ ss) alors
    $\beta_{ref} = Ref_{Ref}(\beta_{in},\beta_{out})$
    E_{ref_out} = COMPUTE_SZ_CONST(β_{ref},β_{out})
    $E_{sol} = \{[[0 \le sol]], [[sol \le 1]]\}$

Exécution d'un appel de procédure

Le succès de l'exécution d'un appel de procédure dépend de deux choses : s'il s'agit d'un appel récursif à la procédure qui est exécutée, il faut tout d'abord vérifier si la taille des termes utilisés comme arguments décroît par rapport à la taille des mêmes termes au début de l'exécution de la procédure. Si ce n'est pas le cas, la terminaison de la procédure n'est pas vérifiée et on sait que cette procédure ne retournera aucun résultat. Si c'est le cas, il faut vérifier que les termes associés aux arguments respectent les conditions définies par au moins un behaviour défini pour la procédure. Si ce n'est pas le cas, le programme ne répond pas aux spécifications, sinon l'exécution de l'appel récursif réussi et on considère que les sorties décrites dans le behaviour trouvé constituent le résultat de cette exécution.

Dans le cas d'un appel non récursif, on applique la même vérification de l'existence d'un behaviour valide pour la procédure appelée.

L'implantation d'une telle exécution est reprise dans l'opération AnalyseClause.

EXTENSION D'UNE SÉQUENCE ABSTRAITE AUX VARIABLES D'UNE CLAUSE APRÈS UNIFICATION

$EXTG\_SA(l, B_1, B_2) = B'$.

Cette opération est utilisée après l'exécution du littéral l et calcule l'effet de cette exécution (qui est donné par la séquence abstraite B_2) sur la séquence abstraite B_1.

Soient :
l un littéral de la forme $X_{i_1}=X_{i_2}$ (n=2) ou $X_{i_1}=f(X_{i_2},...,X_{i_n})$ (f est un symbole de fonction) ou $q(X_{i_1},...,X_{i_n})$ (p est un symbole de prédicat p);
$\{X_{i_1},...,X_{i_n}\} \subseteq \{X_1,...,X_m\}$;
$B_1 = \langle \beta_{1,in}, \beta_{1,ref}, \beta_{1,out}, E_{1,ref\_out}, E_{1,sol} \rangle$;
$dom_{out}(B_1) = \{X_1,...,X_m\}$;
$B_2 = \langle \beta_{2,in}, \beta_{2,ref}, \beta_{2,out}, E_{2,ref\_out}, E_{2,sol} \rangle$;
$dom_{out}(B_2) = \{X_1,...,X_n\}$ $(n \leq m)$;
$B' = \langle \beta_{in}', \beta_{ref}', \beta_{out}', E_{ref\_out}', E_{sol}' \rangle$;
$RESTRG\_SA(l,B_1) \leq input(B_2)$.

L'implantation est :

$\beta'_{in} = \beta_{1,in}$
$\beta_{inter} = Ref_{inter}(l, \beta_{1,out}, \beta_{2,ref})$ $(\Rightarrow \beta_{inter} \leq \beta_{1,out})$
$\beta'_{ref} = Ref_{ref}(\beta_{1,ref}, \beta_{inter})$ $(\Rightarrow \beta'_{ref} \leq \beta_{1,ref})$
$\beta'_{out} = EXTG\_SUBST(l, \beta_{inter}, \beta_{2,out})$
$E^{*1}_{ref\_out} = Ref_{ref\_out}(\beta_{1,ref}, \beta'_{ref}, E_{1,ref\_out}, \beta_{1,out}, \beta_{inter})$
$E^{*2}_{ref\_out} = Refl_{ref\_out}(l, \beta_{2,ref}, \beta_{inter}, E_{2,out}, \beta_{2,out}, \beta_{out}')$
$E'_{ref\_out} = Combine_{out}(E^{*1}_{ref\_out}, E^{*2}_{ref\_out})$
$E^{*1}_{sol} = Ref_{sol}(\beta_{1,ref}, \beta_{ref}', E_{1,sol})$
$E^{*2}_{sol} = Refl_{sol}(l, \beta_{2,ref}, \beta_{inter}, E_{sol}2)$
$E'_{sol} = Combine_{sol}(E^{*1}_{ref\_out}, E^{*1}_{sol}, E^{*2}_{sol})$.

EXTENSION D'UNE SUBSTITUTION AUX VARIABLES D'UNE CLAUSE APRÈS UNIFICATION

EXTG_SUBST(l,β_1,β_2) = β'.

Soient :
$\beta_1 = \langle sv_1, mo_1, ty_1, frm_1, ps_1, E_1 \rangle$ définie sur $I_{p_1}=\{1,...,p_1\}$;
$dom(\beta_1) = \{X_1,...,X_m\}$;
l un littéral de la forme $X_{i_1}=X_{i_2}$ (k=2) ou $X_{i_1}=f(X_{i_2},...,X_{i_k})$ (f est un symbole de fonction) ou $q(X_{i_1},...,X_{i_k})$ (p est un symbole de prédicat p);
$D = \{X_{i_1},...,X_{i_k}\}$ la séquence des variables présentes dans l ;
$D \subseteq dom(\beta_1)$;
$\beta_2 = \langle sv_2, mo_2, ty_2, frm_2, ps_2, E_2 \rangle$ définie sur $I_{p_2}=\{1,...,p_2\}$;
$\beta_2 = l\ \beta_{restr}$ (où β_{restr} = RESTRG_SUBST(l, β_1)) $\Rightarrow \beta_2 \leq \beta_1$ (β_2 est le résultat de l'exécution de l avec β_1);
$dom(\beta_2) = \{X_1,...,X_k\}$ ($k \leq m$);
$\exists$ un mapping t : D $\rightarrow$ $dom(\beta_2)$.

L'implantation se déroule en plusieurs parties :

1) On construit une substitution unique β_{tmp} qui contient les deux substitutions β_1 et β_2.

$\beta_{tmp} = \langle sv_{tmp}, mo_{tmp}, ty_{tmp}, frm_{tmp}, ps_{tmp} \rangle$ définie sur $I_{p_{tmp}}$ (où $p_{tmp}=p_1+p_2$) ;
$\forall$ i : $1 \leq i \leq m$: $sv_{tmp}(X_i) = sv_1(X_i)$
$\forall$ i : $1 \leq i \leq k$: $sv_{tmp}(X_{i+m}) = sv_2(X_i)+p_1$
$\forall$ i : $1 \leq i \leq p1$: $mo_{tmp}(i) = mo_1(i)$
$ty_{tmp}(i) = ty_1(i)$
$\forall$ i : $1 \leq i \leq p2$: $mo_{tmp}(i+p_1) = mo_2(i)$
$ty_{tmp}(i+p_1) = ty_2(i)$
$frm_{tmp} = frm_1 \cup \{ \langle i+p_1, f(i_1+p_1,...,i_n+p_1) \rangle : \langle i, f(i_1,...,i_n) \rangle \in frm_2\}$
$ps_{tmp} = ps_1 \cup \{ (i+p_1, j+p_2) : ps_2(i,j)\}$

2) L'idée est : « On unifie les variables contenues dans β_1 et présentes dans le littéral l avec leur variable correspondante dans β_2. Comme $\beta_2 \leq \beta_1$, tous les termes associés aux variables de β_2 sont $\leq$ que les termes associés aux variables correspondantes dans β_1. Les unifications auront donc pour conséquence de rendre les termes associés aux variable de β_1 et présentes dans l égaux aux termes associés aux variables correspondantes dans β_2. » Comme β_2 est le résultat de l'unification, non seulement on fixe les nouvelles valeurs des variables présentes dans l, mais on propage aussi les effets de l'unification sur les termes liés à ces variables. Pour effectuer cette série d'unification, on construit d'abord la liste des termes à unifier puis on lance l'unification de chacune de ces paires dans β_{tmp}.

List = $\{(sv_1(X_{i_1}),sv_2(X_1)),...,(sv_1(X_{i_n}),sv_2(X_n))\}$
β' = UNIF_LIST(List, β_{tmp})

RESTRICTION D'UNE SÉQUENCE ABSTRAITE AUX VARIABLES D'UNE CLAUSE APRÈS UNIFICATION

RESTRC_SA(c,B) = B'.

Soient :
c une clause ;

$B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$;
$dom_{out}(B)$ = ensemble des variables présentes dans c ;
$B' = \langle \beta_{in}', \beta_{ref}', \beta_{out}', E_{ref\_out}', E_{sol}' \rangle$;
$dom_{in}(B') = dom_{out}(B') = dom_{in}(B)$ = ensemble des variables de tête de c.

L'implantation est :

$\beta_{in}' = \beta_{in}$
$\beta_{ref}' = \beta_{ref}$
β_{out}' = RESTRC_SUBST(c, β_{out}) ($\Rightarrow \exists\ tr_{out} : I_{out}' \rightarrow I_{out}$)
$E_{out}' = (tr_{out} + id)^{<}(E_{ref\_out})$ où id est la fonction identité
$E_{sol}' = E_{sol}$.

RESTRICTION D'UNE SUBSTITUTION ABSTRAITE AUX VARIABLES D'UNE CLAUSE APRÈS EXÉCUTION DE LA CLAUSE.

RESTRC_SUBST(c,β) = β'.

Soient :
c une clause contenant toutes les variables $D = \{X_1,...,X_m\}$;
$\beta = \langle sv,mo,ty,frm,ps,E \rangle$ définie sur $I_p = \{1,...,p\}$;
dom(β) = D ;
$\beta' = \langle sv',mo',ty',frm',ps',E' \rangle$ définie sur $I_{p'}$;
$dom(\beta') = \{X_1,...,X_n\}$ = ensemble des variables de la tête de c = D' ;
$D' \subseteq D$;
$n \leq m$;

L'implantation de cette opération procéde en plusieurs étapes :

1) On a tout d'abord besoin de calculer l'ensemble des index atteignables dans β à partir des variables présente dans la tête de c. Pour cela, on définit la fonction « reachable » qui retourne l'ensemble des index qui composent un index i dans un composant frm.

Reachable(i,frm) = {i} si frm(i)=undef ;

$= \{i\} \cup \bigcup_{j=1}^{n} reachable(i_j, frm)$ si $frm(i)=f(i_1,...,i_n)$.

Et on calcule :

$$I = \bigcup_{X \in D'} reachable(sv(X), frm)$$

2) Ensuite, on établit un mapping entre les index ∈ I et les index de β' :

$t : I \rightarrow I_{p'} = \{1,...,\#I\}$.

3) On définit aussi la fonction partielle tr établissant le mapping inverse de t.

$tr : I_{p'} \rightarrow I_p : \forall\ i \in I : tr(t(i))=i$

4) Enfin, on exécute l'opération de projection à proprement parler :

$\beta' = \langle sv',mo',ty',frm',ps' \rangle$
$sv'(X) = t(sv(X))\ \forall\ X \in D'$
$\forall\ i \in I$: $mo'(t(i)) = mo(i)$
$ty'(t(i)) = ty(i)$
$frm' = \{\langle t(i), f(t(i_1),...,t(i_n))\rangle : \langle i,f(i_1,...,i_n)\rangle \in frm \wedge i \in I\}$

ps'= {(t(i),t(j)) : i,j ∈ I ∧ ps(i,j)}
E' = $tr^{<}$(E)

INTERPRÉTATION DES RÉSULTATS CALCULÉS SUR UNE CLAUSE

Procédure AnalyseClause

AnalyseClause(β_{in}, c, se) = ⟨ success, B_{out} ⟩.

Soient :
β_{in} une substitution abstraite en entrée ;
c une clause de la forme $p(X_1,...,X_n)$:- $l_1,...,l_s$;
se une size expression ;
success un booléen ;
B_{out} une séquence abstraite en sortie.

Si S est une séquence finie, on peut prouver que l'exécution de chaque appel récursif $l_i=p(X_{i_1},...,X_{i_n})$ se termine en utilisant se :

Si $\theta \in Cc(\beta_{in})$,
$\langle \theta, l_1,...,l_{i-1} \rangle \rightarrow S'$, S' est le résultat de l'exécution
des littéraux $l_1,...,l_{i-1}$ avec θ,
$l_i=p(X_{i_1},...,X_{i_n})$ un appel récursif à la procédure en exécution p,
$\theta' \in S'$
Alors [[se]] $\langle ||X_1\theta||,...,||X_n\theta|| \rangle >$ [[se]] $\langle ||X_{i_1}\theta'||,...,||X_{i_n}\theta'|| \rangle$.

Cette condition dit que la taille des variables au début de l'exécution de la clause est supérieure à la taille des variables correspondantes utilisées lors de l'exécution de l_i.

L'implantation est :

β_{in} = input(B)
B_0 = EXTC_SA(c,β_{in})
∀ k ∈ {1,...,s} où c ≡ $p(X_1,...,X_n)$:-$l_1,...,l_s$
$\beta_{k,inter}$ = RESTRG_SA(l_k,B_{k-1})
si $l_k \equiv X_{i_1}=X_{i_2}$ alors $B_{k,aux}$ = AI_VAR_SA($\beta_{k,inter}$)
si $l_k \equiv X_{i_1} = f(X_{i_2},...,X_{i_m})$ alors $B_{k,aux}$ = AI_FUNC_SA($\beta_{k,inter}$, f)
si $l_k \equiv q(X_{i_1},...,X_{i_m}) \wedge q \neq p$ alors
⟨ $success_k$, $B_{k,aux}$ ⟩ = LOOKUP_SA($\beta_{k,inter}$, q, sat)
si $l_k \equiv q(X_{i_1},...,X_{i_m}) \wedge q = p$ alors
⟨ $success_k$, $B_{k,aux}$ ⟩ = LOOKUP_SA($\beta_{k,inter}$,q,sat)
$success_k$ = $success_k$' ∧ CHECK_TERM(l_k,B_{k-1},se)
B_k = EXTG_SA(lk, B_{k-1}, $B_{k,aux}$)
Si ∃ k : $((l_k \equiv q(X_{i_1},...,X_{i_m}) \wedge \neg success_k) \vee$
$(l_k \equiv p(X_{i_1},...,X_{i_n}) \wedge (\neg successk \vee \beta_{k,inter} > \beta_{in}))$
alors success = false
sinon success = true, B_{out} = RESTRC(c, B_{out}).

INTERPRÉTATION DES RÉSULTATS CALCULÉS SUR UNE PROCÉDURE

Procédure AnalyseProcedure

AnalyseProcedure (B,p,se) = success.

Soient :
B une séquence abstraite ;
p un nom de procédure du programme ;
se une size expression ;
success une valeur booléenne.

Si la procédure p est constituée des clauses $c_1,...,c_r$, l'implantation est :

$\forall k \in \{1,...,r\} \Rightarrow (success_k, B_k) = AnalyseClause(B,c_k,se)$
si $\exists k \in \{1,...,r\} : \neg success_k$ alors success = false
sinon $B' = CONC\_SA(B_1,...,B_r)$, success = $(B' \leq B)$.

INTERPRÉTATION DES RÉSULTATS CALCULÉS SUR UN PROGRAMME.

AnalyseProgram(P,Sbeh) = success.

Soient :
P un programme ;
Sbeh un ensemble de behaviours de la forme $\langle p, \{\langle B_1,se_1\rangle,...,\langle B_q,se_q\rangle\}\rangle$;
success un booléen ;
il y a un behaviour dans Sbeh pour chaque procédure p définie dans P.

L'implantation est :

Lecture et construction des représentations internes du programme Prolog P et des spécifications de behaviours Sbeh ;
sat = MAKESAT(Sbeh) ;
$\forall p \in P : \forall \langle B,se\rangle \in Beh_p : success_{\langle p,B,se\rangle} = AnalyseProcedure(B,p,se)$;
success = $(\forall p \in P : \forall \langle B,se\rangle \in Beh_p : success_{\langle p,B,se\rangle})$.

OPÉRATIONS D'UNIFICATION GÉNÉRALES

Opération UNIF

UNIF(i,j,β) = ⟨β',tr',ss',sf',U' ⟩.

Cette opération unifie les termes i et j dans la substitution abstraite β. Les résultats sont une substitution abstraite β' dans laquelle les termes ont été unifiés, un structural mapping tr' de β sur β', deux valeurs booléennes ss (sure success) et sf (surefailure)

indiquant si l'unification des termes réussit toujours ou ne réussit jamais, et U l'ensemble des indices de β' qui n'ont pas été affectés par l'unification.

Soient :
β = ⟨sv, mo, ty, frm, ps, E ⟩ définie sur I ;
β = ⟨sv', mo', ty', frm', ps', E' ⟩ définie sur I' ;
dom(β)=dom(β')={X1,...,Xn} ;
tr' : I→I' ;
ss, sf deux valeurs booléennes ;
U' l'ensemble des indices non affectés par les unifications.

L'implantation est :

⟨ β', tr', ss', sf', U' ⟩ = UNIFLIST({(i,j)}, β).

Opération UNIFLIST

UnifList(List,β). = ⟨β',tr',ss,sf,U'⟩.

Cette opération unifie les termes i et j contenus dans chaque paire de la liste List dans la substitution abstraite β. Les résultats sont une substitution abstraite β' dans laquelle les termes ont été unifiés, un structural mapping tr' de β sur β', deux valeurs booléennes ss (sure success) et sf (surefailure) indiquant si l'ensemble des paires de termes unifiés réussit toujours ou ne réussit jamais, et U l'ensemble des indices de β' qui n'ont pas été affectés par les unifications.

Soient :
β = ⟨sv, mo, ty, frm, ps, E ⟩ définie sur I ;
β = ⟨sv', mo', ty', frm', ps', E' ⟩ définie sur I' ;
dom(β)=dom(β')={X1,...,Xn} ;
tr' : I→I' ;
ss, sf deux valeurs booléennes ;
list = {(i,j) | i,j ∈ I} ;
U' l'ensemble des indices non affectés par les unifications.

L'implantation est :

$\beta^0 = \beta$
list = $\{(i_1,j_1),\ldots,(i_m,j_m)\}$
∀ k : 1≤k≤m :
  si $frm^{k-1}(i_k)=frm^{k-1}(j_k)$=undef alors
    $\langle \beta^k, ss^k \rangle$ = UACT1(i_k,j_k,β^{k-1})
    sf^k = false
  si $frm^{k-1}(i_k)$ = undef et $frm^{k-1}(j_k)$=f(...) alors
    $\langle \beta^k, ss^k \rangle$ = SPECAT(i_k,j_k,β^{k-1})
    sf^k = false
  si $frm^{k-1}(ik)$ = f(...) et $frm^{k-1}(j_k)$=undef alors
    $\langle \beta^k, ss^k \rangle$ = SPECAT(j_k,i_k,β^{k-1})
    sf^k = false
  si $frm^{k-1}(i_k)=f(v_1,\ldots,v_n)$ et $frm^{k-1}(j_k)=f(u_1,\ldots,u_n)$ alors
    $\beta^{k,0}=\beta^{k-1}$
    ∀ l : 1≤l≤n : $\beta^{k,l}$ = FCTA($\beta^{k,l-1}$, v_l, u_l)
    $\beta^k = \beta^{k,l}$
    sf^k = false, ss^k = true
  sinon stop avec échec (sf^k = true, ss^k = false)

$\beta' = Normalize(\beta^m)$
$ss = (\forall\ k : 1 \le k \le m : ss^k)$
$sf = (\exists\ k : 1 \le k \le m : sf^k)$
tr' = structural mapping entre β et β'
$U' = ? ?$

Opérations UACT1

UACT1(i, j, β) = $\langle\ \beta'$, success $\rangle$.

Cette opération était définie dans GAIA. Elle unifie les termes i et j, tous deux n'ayant pas de forme définie, dans la substitution β.

Soient :
$\beta = \langle$ sv,mo,ty,frm,ps,E $\rangle$ définie sur $I_p = \{1,...,p\}$;
$i, j \in I_p$;
frm(i) = frm(j) = undef ;
$\beta' = \langle$ sv',mo',ty',frm',ps',E' $\rangle$ définie sur $I_{p'} = \{1,...,p'\}$;
$T = \langle\ t_1,...,t_m\ \rangle$ un m-uple de termes.

L'opération respecte la condition suivante :

$$\sigma \in mgu(t_i,t_j) \wedge T \in Cc(\beta) \Rightarrow T\sigma \in Cc(\beta')$$

où σ est une substitution concrète.

L'implantation de l'opération est :

1) On définit la condition SH $\Leftrightarrow$ ps*(i,k) $\vee$ ps*(j,k) qui est vérifiée si les termes i et k ou j et k sont liés de façon quelconque ;

2) Ensuite :

mo'(k) =	mo(k)	si SH est faux ;
	UAT_{mode}(mo(i),mo(j))	si SH est vrai et (k=i $\vee$ k=j) ;
	$MATCH_{mode}$(mo(k),f,mo(k_1),...,mo'(k_n))	si SH est vrai et $i \neq k \neq j$ et que frm(k)=f(k_1,...k_n) ;
	IAT_{mode}(mo(k))	sinon ;
ty'(k) =	ty(k)	si SH est faux ;
	UAT_{type}(ty(i),ty(j),mo(i),mo(k))	si SH est vrai et (k=i $\vee$ k=j) ;
	$MATCH_{type}$(ty(k), f, ty'(k_1),...,ty'(k_n),mo'(k))	si SH est vrai et $i \neq k \neq j$ et que frm(k)=f(k_1,...k_n) ;
	IAT_{type}(ty(k))	sinon ;
frm' =	frm ;	
ps_1 =	{(k,l) | ps(k,l) $\wedge$ $\neg$ground(mo'(k)) $\wedge$ $\neg$ground(mo'(l))}	
ps_2 =	{(k,l) | $\exists$ k',l' : ps(k',k) $\wedge$ ps(l',l) $\wedge$ k',l' $\in$ {i,j}}	
ps' =	ps_1	si ground(mo'(i))
	$ps_1 \cup ps_2$	sinon ;

FCTA(i,j,β).

Opération Specat

SPECAT(i, j, β) = β'.

Cette opération était définie dans GAIA. Elle est utilisée pour unifier deux termes t_i et t_j dans la substitution β, et dans laquelle frm(t_i)=undef et frm(t_j)=f(j_1,...,j_n). Une telle

unification est réalisée en deux étapes : on unifie d'abord t_i à un terme $f(Y_1,...Y_n)$ créé de toutes pièces (Y_k sont des nouveaux termes variables distincts), ensuite, on unifie un à un les j_k et Y_k. La création de $f(Y_1,...,Y_n)$ unifié à t_i est réalisé par cette opération, l'unification des Y_k et t_k est réalisée par la procédure d'unification d'une liste de paires de termes.

Soient :
$\beta=\langle$ sv,mo,ty,frm,ps,E $\rangle$ définie sur I_p ;
$i, j \in I_p$;
frm(i) = undef ;
frm(j) = f(j_1,...,j_n)
$\beta'=\langle$ sv',mo',ty',frm',ps',E' $\rangle$ définie sur $I_{p'}$;
$T = \langle t_1,...,t_p \rangle$ un p-tuple de termes ;
$Y_1,...,Y_n$ sont de nouvelles variables distinctes non présentes dans T.

L 'opération respecte la condition suivante :

Si $T \in Cc(\beta) \wedge$
t_i, t_j sont unifiables $\wedge$
$\sigma \in mgu(f(Y_1,...,Y_n), t_i)$
Alors $\langle t_1,...,t_p,Y_1,...,Y_n\rangle\sigma \in Cc(\beta')$.

L'implantation procède en plusieurs étapes :

1) On extrait d'abord la meilleure approximation des modes et des types des termes composant f. On calcule aussi l'ensemble des termes liés au terme j à l'aide de la fonction « reachable ».

$\langle M_1,...,M_n \rangle = EXTR_{mode}(mo(i),f)$
$\langle T_1,...,T_n \rangle = EXTR_{type}(ty(i),f,mo(i),M_1,...,M_n)$
J = reachable(j,frm)

2) Si (i) les types T_k sont $\perp$ et que le type de i était défini comme « list » ou comme $\perp$, ou bien (ii) si on tente d'unifier le terme i au terme j alors que i est un composant de j, le résultat est $\perp$.

$\beta' = \perp$ si $(\{\perp\} = \{T_1,...,T_n\} \wedge \neg(anylist \leq ty(i))) \vee i \in J$

3) Dans le cas contraire on calcule β' :

$\beta'= \langle$ sv',mo',ty',frm',ps',E' $\rangle$
$\forall$ k : $1 \leq k \leq p$:
  si $\neg ps^*(i,k) \vee k \in$ reachable(j,frm) alors
    mo'(k) = mo(k)
  sinon si k = i alors
    si mo(i) $\geq$ var alors
      $mo'(k)=LUB_{mode}(MATCH_{mode}(mo(i),M_1,...,M_n),$
        $CONS_{mode}(f,var,...,var))$
    sinon
      mo'(k) = mo(i)
  sinon si $k \neq i \wedge$ frm(k)=g(k_1,...,k_m) alors
    $mo'(k)=MATCH_{mode}(mo(k),g,mo'(k_1),...,mo'(k_m))$
  sinon si $k \neq i \wedge$ frm(k) = undef alors
    si mo(i) $\geq$ var alors
      $mo'(k)=IAT2_{mode}(mo(k),CONS_{mode}(f,var,...,var))$
    sinon
      mo'(k) = mo(k)
$\forall$ k : $1 \leq k \leq p$:
  si k = i alors

$T = MATCH_{type}(ty(i),f,T_1,...,T_n,mo(i))$
si $anylist \leq ty(i)$ alors
$M = CONS_{mode}(f,var,...,var)$
$ty_1(k) = LUB_{type}(T,CONS_{type}(M,f,anylist,...,anylist))$
sinon
$ty_1(k) = \top$
si $(k \neq i \wedge frm(k)=g(k_1,...,k_m))$ alors
$ty_1(k) = MATCH_{type}(ty(k),g,ty_1(k_1),...,ty_1(k_m),mo'(k))$
si $(k \neq i \wedge frm(k)=undef)$ alors
si $anylist \leq ty(i)$ alors
$M=CONS_{mode}(f,var,...,var)$
$ty_1(k)=IAT2_{type}(mo(k),mo'(k),ty(k),CONS_{type}(M,f,anylist,...,anylist))$
sinon
$ty_1(k) = ty(k)$
si $(\neg ps^*(i,k) \vee k \in reachable(j,frm))$ alors
$ty'(k) = GLB(ty(k), ty_1(k))$
sinon
$ty'(k) = ty_1(k)$
$\forall k : p < k \leq p+n :$
si $mo(i) \geq var$ alors
$mo'(k) = LUB_{mode}(M_{k-p},var)$
sinon
$mo'(k) = M_{k-p}$
si $anylist \leq ty(i)$ alors
$ty'(k) = LUB(T_{k-p}, anylist)$
sinon
$ty'(k) = T_{k-p}$
$frm' = frm \cup \{ \langle i,f(p+1,...,p+n) \rangle \}$
$ps' = (ps / symmetric(ps_1)) \cup symmetric(ps_2) \cup ps_3 \cup ps_4$
où :
$symmetric(ps_n)$ est la relation symétrique de ps_n ;
$ps_1 = \{(i,k) \mid 1 \leq k \leq p, ps(i,k)\}$;
$ps_2 = \{(k,p+l) \mid 1 \leq k \leq p, ps(i,k), 1 \leq l \leq n, mo'(k) \neq ground \neq mo'(p+l)\}$;
$ps_3 =$ $\{(k,k) \mid p<k \leq p+n\}$ si $mo(i) \geq var$;
$\varnothing$ sinon ;
$ps_4 =$ $\varnothing$ si $mo(i)=var$;
$\{(k,l) \mid p<k,l \leq p+n, M_{k-p} \neq ground \neq M_{l-p}\}$ sinon.

« L'implantation est basée sur le raisonnement de l'unification de t_i et $f(y_1,...,y_n)$, en supposant que t_i et t_j sont unifiables.

On a fait l'hypothèse que t_j est un terme de la forme $f(t_{j_1},...,t_{j_n})$. S'il est unifiable avec le terme t_i, c'est que ce terme t_i est soit une variable, soit un terme de la forme $f(u_1,...,u_n)$.

Si le terme t_i est une variable y, ce cas n'est possible que si $var \leq mo(i)$. La substitution concrète $\sigma=\{y \leftarrow f(y_1,...,y_n)\}$ est le mgu de l'unification de t_i et t_j. Le mode de t_i devient $CONS_{mode}(f,var,...,var)$, *var* étant le mode des nouvelles variables y_l. Les variables $y_1,...,y_n$ étant de nouvelles variables, elles ne sont pas instanciées et leur mode reste var.

Si le terme t_i est un terme composé de la forme $f(u_1,...,u_n)$, le mgu σ de l'unification de t_i et t_j est $\{y_1 \leftarrow u_1,...,y_n \leftarrow u_n\}$ puisque les y_i sont de nouvelles variables. Les modes des $y_1,...,y_n$ sont les modes M_i des termes u_i donnés par $EXTR_{mode}(mo(i),f)$. Le mode de t_i n'est pas modifié.

Ensuite, pour propager les résultats de cette unification aux autres termes t_k, on fait $t_k\sigma$. $t_k\ \sigma$ diffère de t_k seulement si t_i est une variable y et t_k contient y (dans le cas contraire seuls $y_1,...,y_n$ sont modifiés). Cela ne peut pas se produire si t_k ne partage pas de variable avec t_i, ni si t_k est un terme qui compose t_j (parce que t_j devrait alors contenir y et l'unification échouerait). Dans tous les autres cas, si la forme de t_k est connue, on ajuste sont mode en fonction du nouveau mode des termes le composant. Si la forme de t_k n'est pas connue, t_k devient $t_k\{y \leftarrow f(y_1,...,y_n)\}$. Son nouveau mode est alors $IAT2_{mode}(mo(k),CONS_{mode}(f,var,...,var))$. »

Opération FCTA

FCTA(i,j,β) = β'.

Cette opération ajoute une égalité entre les termes i et j dans β et propage cette égalité à tous les autres termes. Les résultat est la substitution pseudo-abstraite β' dans laquelle toute référence à j a été remplacée par une référence à i.

Soient :
$\beta = \langle sv,mo,ty,frm,ps,E \rangle$ définie sur $I=\{1,...,p\}$;
$dom(\beta) = \{X_1,...,X_n\}$;
$\beta' = \langle sv',mo',ty',frm',ps',E' \rangle$ définie sur $I' = I / \{j\}$;
$i,j \in I$;
$frm(i)=frm(j)$.

L'implantation est :

$sv'(X) = sv(X)\ \forall\ X \in dom(\beta) : sv(X) \neq j$.
$mo'(k) = mo(k)\ \forall\ k \in I / \{j\}$;
$ty'(k) = ty(k)\ \forall\ K \in I / \{j\}$;
$frm' = \{\langle k,f(k_1',...,k_n') \rangle \mid k \in I / \{j\} \wedge \langle k,f(k_1,...,k_n) \rangle \in frm \wedge ((k_i'=k_i \wedge k_i \neq j) \vee (k_i'=i \wedge k_i=j))\}$;
$ps' = \{(k,l) \mid k,l \in I/\{j\},\ ps(k,l)\} \cup \{(i,k) \mid k \in I,\ ps(j,k)\} \cup \{(k,i) \mid k \in I,\ ps(k,j)\}$.

E' = ?

Opération LOOKUP_SA

LOOKUP_SA(β, p, sat) = ⟨success, B⟩.

Cette opération recherche dans sat_p la séquence abstraite B la plus précise dont la substitution en entrée est au moins aussi générale que β. Si B existe, l'opération retourne success=vrai et cette séquence. Sinon, success est faux et la séquence retournée est indéfinie.

Soient :
$\beta = \langle sv,mo,ty,frm,ps,E \rangle$;
$dom(\beta) = \{X_1,...,X_n\}$;
$p \in P$;
$sat = (sat_p)_{p \in P}$ où $\forall\ p \in P$ est un symbole de prédicat d'arité n et sat_p est un ensemble fini de séquences abstraites ;
success est un booléen ;
$B = \langle\ \beta_{in},\beta_{ref},\ \beta_{out},\ E_{ref\_out},\ E_{sol}\ \rangle$;
$dom_{in}(B) = dom_{out}(B) = dom(\beta)$.

L'opération vérifie la condition :

$$\exists\ B' \in sat_p : \beta \leq input(B') \Rightarrow success = true \wedge B = min\ \{B' \mid B' \in sat_p,\ \beta \leq input(B')\}.$$

Opération CHECK_TERM

CHECK_TERM(l,B,se) = term.

Cette opération est utilisée lorsqu'on exécute un appel récursif d'une procédure p. Elle vérifie la terminaison de cet appel récursif en se basant sur l'état de la séquence au moment de l'appel et sur une size expression.

Soient :
l un littéral de la forme $p(X_{i_1},...,X_{i_n})$ ($p \in P$ est un symbole de prédicat d'arité n) ;
l est un appel récursif ;
$B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$;
$dom_{in}(B) = \{X_1,...,X_n\}$;
$dom_{out}(B) = \{X_1,...,X_m\} \supseteq \{X_{i_1},...,X_{i_n}\}$ ($n \leq m$) ;
$sv_{ref} : \{X_1,...,X_n\} \rightarrow I_{ref}$;
$sv_{out} : \{X_1,...,X_m\} \rightarrow I_{out}$;
$in_{ref} : I_{ref} \rightarrow I_{ref} + I_{out}$ est une injection ;
$in_{out} : I_{out} \rightarrow I_{ref} + I_{out}$ est une injection ;
$t : \{X_1,...,X_n\} \rightarrow \{X_{i_1},...,X_{i_n}\}$ telle que $t(X_j) = X_{i_j}\ \forall\ j \in \{1,...,n\}$;
$sz_{ref\_out} : I_{ref} + I_{out} \rightarrow \{sz(i) : i \in I_{ref} + I_{out}\}$ telle que $sz_{ref\_out}(i)=sz(i)$;
se est une expression linéaire positive de l'ensemble des expressions définies sur $X_1,...X_n$;
$[[se]] : \aleph^n \rightarrow \aleph$;
term est un booléen.

L'opération respecte la condition :

$$term = vrai \Rightarrow \forall\ \langle \theta, S \rangle \in Cc(B) : \forall\ \theta' \in Subst(S) :$$
$$[[se]]\ \langle\ ||X_1\theta||,...,||X_n\theta||\ \rangle > [[se]]\ \langle\ ||X_{i_1}\theta'||,...,||X_{i_n}\theta'||\ \rangle .$$

Cette condition dit que la taille des variables au début de l'exécution de la clause est supérieure à la taille des variables correspondantes utilisées lors de l'exécution de l_i.

L'implantation est :

term = false
$E = \hat{in}_{ref}(E_{ref}) \cup \hat{in}_{out}(E_{out}) \cup E_{ref\_out} \cup \{(\hat{sz}_{ref\_out} \circ \hat{in}_{ref} \circ \hat{sv}_{ref})(se) \leq (\hat{sz}_{ref\_out} \circ \hat{in}_{out} \circ \hat{sv}_{out} \circ \hat{t})(se)\}$
si EMPTY(E) then term = true

Annexe E : Implantation des opérations abstraites

OPÉRATIONS SUR LES MODES

Construction de modes

Soient :
$M_1,...,M_n \in$ Modes les modes des termes $t_1,...,t_n$;
f un symbole de fonction n-aire.

Implémentation : Soit C la condition $\exists\, i : 1 \leq i \leq n : M_i = \bot$.

M' = $\bot$ si C ;
ground si $\forall\, i : 1 \leq i \leq n : M_i$ = ground ;
ngv si $\neg C \wedge \exists\, i : 1 \leq i \leq n : M_i \leq$ noground ;
novar sinon.

S'il existe un mode Mi = $\bot$, la concrétisation de ce mode ne renvoie aucun terme. On ne sait dans pas construire de terme composé et M' = $\bot$.

Si tous les termes composant f sont clos, leur concrétisation ne générera que des termes clos. On ne sait construire que des termes composés clos. Donc M'=ground.

Si certains termes composant f n'est pas clos, le mode de f dépend de la précision que l'on a sur ces termes. Si on sait que, par exemple, un terme composant f est une variable, on peut affirmer que f ne sera ni clos (puisqu'il contient une variable) ni une variable (puisqu'il est composé). M' vaut alors ngv. De même si on sait que ce terme n'est ni une variable ni un terme clos, ou bien si on sait seulement qu'il n'est pas un terme clos (on ignore s'il est une variable).

Enfin, si on sait seulement que les termes de f ne sont sûrement pas des variables (ils peuvent être éventuellement clos), on ne peut pas affirmer que f ne sera jamais clos. L'approximation la plus précise que l'on peut faire est de dire que f n'est pas une variable. Donc, M' = novar.

Extraction de modes

$EXTR_{mode}(M,f) = (M_1,...,M_n)$.

Soient :
M $\in$ Modes ;

f un symbole de fonction n-aire ;
$t_1,...,t_n$ des termes.

Implantation : $\forall\ 1 \leq i \leq n$:

$M_i =$ $\perp$ si $M \in \{\perp, var\}$;
ground si $M \in \{ground, gv\}$;
noground si $M \in \{noground, ngv\} \wedge n = 1$;
any sinon.

Le cas où le mode de f est var est un cas impossible, f étant un terme composé. On reporte une erreur sous la forme de $\perp$.

Si le mode de f est clos, la concrétisation de f donne des termes composés clos, et donc des termes composants clos.

Si le mode de f est ngv et que f est composé d'un seul terme, la concrétisation du mode donne l'ensemble des termes composés d'un terme qui ne sont ni variable ni clos. Le terme composant est donc non clos.

Si le mode de f est ngv et que f est composé d'au moins deux termes, la concrétisation peut générer entre autres des termes de la forme f(a,X), f(Y,b), f(X,Y) ou encore f([a|X],Y). Les termes composants pouvant être n'importe quoi, leur mode est any.

Si le mode de f est ng et que f est composé d'un seul terme, les termes générés par la concrétisation sont soit des variables (or c'est impossible) soit des termes ni clos ni variable. On se ramène donc au cas M=ngv et n=1.

Si le mode de f est ng et f est composé d'au moins deux termes, les termes générés par la concrétisation sont soit des variables (or c'est impossible) soit des termes ni clos ni variable. On se ramène donc au cas M=ngv et n>1.

Si le mode de f est gv, les termes composés générés par la concrétisation sont soit des variables (or, c'est impossible), soit des termes clos. On se ramène au cas où M=ground.

Si le mode de f est nv, on peut seulement affirmer que la concrétisation du mode génère des termes composés qui ne sont pas des variables. Or c'est toujours le cas. Comme on ne sait rien affirmer de plus, on considère le mode le plus général pour les termes composant f.

Matching de modes

$MATCH_{mode}(M, f, M_1,..,M_n) = M'$.

Soient :
M le mode de f ;
$M_1,...,M_n \in$ Modes les nouveaux modes des termes composant f ;
f un symbole de fonction n-aire.

Implantation :

$m_1 =$ $CONS(f,M_1,...,M_n)$ si $M \neq \perp \wedge M \neq$ ground ;
$\perp$ sinon ;
$m_2 =$ ground si $M \geq$ ground $\wedge\ \forall\ i : M_i \geq$ ground ;
$\perp$ sinon ;
$M' = LUB(m_1,m_2)$.

Si le mode actuel de f est clos ou $\perp$, il ne peut pas avoir été instancié à une autre valeur. On retourne une erreur sous la forme de $\perp$. Si ce n'est pas le cas, on recalcule le nouveau mode de f à partir des modes de ses composants. Soit m_1 le nouveau mode de f.

Soit m_2 le résultat de ces deux conditions. Si une de ces deux conditions n'est pas respectée, on donne la valeur $\perp$ à m_2 afin de reporter une erreur à l'aide de l'opération LUB. Sinon, on donne la valeur ground à m_2 pour que le LUB assigne la valeur de m_1 à M'.

Unification abstraite de modes

$UAT(M_1,M_2) = M'$.

Soient :
$M_1,M_2,M' \in$ Modes ;
t_1,t_2 deux termes ;
σ une substitution abstraite.

L'implantation est représentée par le tableau (comme l'opération est symétrique, M1 et M2 peuvent être lu verticalement ou horizontalement):

	⊥	var	ngv	ground	noground	gv	novar	any
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
var	⊥	var	ngv	ground	noground	gv	novar	any
ngv	⊥	ngv	novar	ground	novar	novar	novar	novar
ground	⊥	ground	ground	ground	ground	ground	ground	ground
noground	⊥	noground	novar	ground	any	any	novar	any
gv	⊥	gv	novar	ground	any	gv	novar	any
novar	⊥	novar	novar	ground	novar	novar	novar	novar
any	⊥	any	novar	ground	any	any	novar	any

Par exemple, la concrétisation d'un mode M_1=noground génère des termes t_1 variable et des termes ni variable ni clos, et la concrétisation d'un mode M_2=ngv génère des termes t_2 ni variable ni clos.
Si on unifie un terme t_1 ni variable ni clos et un terme t_2 ni variable ni clos, on peut obtenir un terme ni variable ni clos ou un terme clos. Si on unifie un terme t_1 variable et un terme t_2 ni variable ni clos, on obtient un terme ni variable ni clos. Les termes résultats sont donc soit ni clos ni variable ou bien clos. C'est à dire l'ensemble des termes générés par novar.

Instanciation abstraite de modes

$IAT_{mode}(M)=M'$.

L'implantation est représentée par le tableau :

M	⊥	var	ngv	ground	noground	gv	novar	any
M'	⊥	any	novar	ground	any	any	novar	any

On voit que, après instanciation, :
- un terme variable peut rester une variable, devenir clos ou bien ni clos ni variable ;
- un terme ni clos ni variable peut rester ni clos ni variable ou bien devenir clos ;
- un terme clos reste clos.

Les autres possibilités sont basées sur ces trois cas.

instanciation abstraite spécialisée de modes

$IAT2_{mode}(M_1,M_2)=M'$.

Soient :
$M_1,M_2 \in$ Modes ;
t_1,t_2 deux termes ;
y une variable.

L'implantation est représentée par le tableau :

		M_2							
		⊥	var	ngv	ground	noground	gv	novar	any
	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	var	⊥	var	noground	gv	noground	gv	any	any
	ngv	⊥	ngv	ngv	novar	ngv	novar	novar	novar
M 1	ground	⊥	ground	ground	ground	ground	ground	ground	ground
	noground	⊥	noground	noground	any	noground	any	any	any
	gv	⊥	gv	any	gv	any	gv	any	any
	novar	⊥	novar	novar	novar	novar	novar	novar	novar
	any	⊥	any	any	any	any	any	any	any

Exemple 1 :
Soient t_1=X, t_2=Z, M_1=M_2=var. Si on exécute $t_1\{X \leftarrow t_2\}$, on obtient t'=Z et M'=var. Si on exécute $t_1\{W \leftarrow Z\}$, on a t'=t_1 et M'=M_1. M' est dans les deux cas égal à var.

Exemple 2 :
Soient t_1=[a|X], t_2=b, M_1=ngv et M_2=ground. Si on exécute $t_1\{X \leftarrow t_2\}$, alors t'=[a|b] et M'=ground. Si on exécute $t_1\{W \leftarrow t_2\}$, on a t'=t_1 et M'=M_1. Donc M'∈{ground,ngv}, c'est à dire, M'=nv.

Exemple 3 :
Soient t_1=f(X,a,c), t_2=[Z|Y], M_1=novar et M_2=noground. Si on exécute $t_1\{X \leftarrow t_2\}$, on a t'=f([Z|Y],a,c) et M'=novar. Si on exécute $t_1\{W \leftarrow t_2\}$, on a t'=t_1 et M'=M_1. Donc M'=novar dans les deux cas.

Opération $Unist_{mode}$

$UNIST_{mode}(M) = M'$.

L'implantation est :

M' = var si M = var.
noground si M ∈ {ngv, noground}
⊥ si M = ⊥
any sinon.

OPÉRATIONS SUR LES TYPES

Les opérations sur les types utilisent les résultats calculés par les opérations sur les modes.

Construction de types

$CONS_{type}(M_1,...,M_n,M',f,T_1,...,T_n) = T'$.

Soient :
$M_1,...,M_n$ les modes des termes composants f ;
M' le nouveau mode de f ;
$T_1,...,T_n$ les types des termes composants f ;
f un symbole de fonction n-aire
$M' = CONS_{mode}(f, M_1,...,M_n)$.

L'implantation définit d'abord la condition :

$C \Leftrightarrow \exists i : 1 \leq i \leq n : T_i = \bot$

Puis calcule T' :

$T' = \bot$ si $C \vee M' = \bot$
list si $(\neg C \wedge f = cons/2 \wedge T_2 = list \wedge n = 2) \vee (f = [] \wedge n = 0)$
anylist si $(\neg C \wedge f = cons/2 \wedge T_2 = anylist)$
any sinon.

Extraction de types

$EXTR_{type}(T,f,M,M_1',...,M_n') = (T_1',...,T_n')$.

Soient :
T le type actuel de f ;
f un symbole de fonction n-aire $f(t_1,...,t_n)$;
M le mode actuel de f ;
$M_1',...,M_n'$ les nouveaux modes des termes $t_1,...,t_n$;
$EXTR_{type}(M,f) = (M_1',...,M_n')$.

L'implantation de l'opération est :

$\forall i : 1 \leq i \leq n$:
$T_i' = \bot$ si $(T = \bot \vee M \in \{\bot, var\} \vee (f \neq cons/2 \wedge T = list))$;
$T_1' =$ any $\wedge\ T_2' = list$ si $(f = cons/2 \wedge T = list)$;
$T_i' =$ anylist si $M_i' = var \vee (f = cons/2 \wedge T = anylist)$;
$T_i' =$ any sinon.

Instanciation abstraite de types

$IAT_{type}(T) = T'$.

L'implantation est :

$T' = \bot$ si $T = \bot$;
$T' = list$ si $T = list$;
$T' = any$ sinon.

Matching de types

$MATCH_{type}(T,f,T_1,...,T_n,M') = T'$.

Soient :
T le type de f ;
M' le mode de f après instanciation ;
f un symbole de fonction n-aire $f(t_1,...,t_n)$;
$T_1,...,T_n$ les nouveaux types des termes $t_1,...,t_n$.

L'implantation est :

$T' = GLB(IAT_{type}(T), CONS_{type}(M',f,T_1,...,T_n))$.

On calcule le nouveau type de f en se basant sur les nouvelles instances des termes le composant et sur son mode après instanciation. On calcule ensuite le nouveau type de f suite à une instanciation arbitraire du terme composé. Enfin, on retient le résultat le plus précis.

L'unification abstraite de types

$UAT_{type}(M_1,M_2,M',T_1,T_2) = T'$.

Soient :
M_1, M_2 les modes de t_1 et t_2 ;
$M' = UAT(M_1,M_2)$;
T_1, T_2 les types de $t_1$et t_2.

L'implantation est :

$T' =$ $\perp$ si $T_1 = \perp \vee T_2 = \perp \vee M' = \perp$;
T_2 si $(M_1 = var \wedge T_2 \neq \perp)$;
list si $(T_1 = list \wedge T_2 \neq \perp)$;
$UAT_{type}(M_1,M_2,M',T_2,T_1)$ si $T_2 < T_1$;
anylist si $M' = var$;
any sinon.

Instanciation abstraite spécialisée

$IAT2_{type}(M_1,M_2,M',T_1,T_2) = T'$.

Soient :
M_1,M_2 le mode de t_1 et de t_2 ;
T_1, T_2 le type de t_1 et de t_2 ;
$M' = IAT2_{mode}(M_1,M_2)$.

L'implantation est :

$T' =$ $\perp$ si $T_1 = \perp \vee T_2 = \perp$;
list si $(T_1 = list \wedge T_2 \neq \perp)$;
T_2 si $(M_1 = var \wedge T_1 \neq \perp \wedge T_2 \neq list)$;
anylist si $(M_1 = var \wedge T_1 \neq \perp \wedge T_2 = list) \vee (M' = var)$;
any sinon

Opération $Uninst_{type}$

$UNInst_{type}(Y) = T'$.

L'implantation est :

$T' =$ anylist si $T \in \{list, anylist\}$
$\perp$ si $T = \perp$
any sinon.

OPÉRATIONS SUR LE COMPOSANT E

Opération Sum_{Sol}

$SUM_{sol}(E_1,E_2) = E'$.

Soient :
$E_k \in Sizes_{I+\{sol\}}$ (k = 1,2)
$E' \in Sizes_{I+\{sol\}}$.

L'implantation est :

sol_1, sol_2 deux nouvelles variables
$E_{sol} = tr^{<}_{sol}(E_1[sol \rightarrow sol_1] \cup E_2[sol \rightarrow sol_2] \cup \{[[sol = sol_1 + sol_2]]\})$

où :
- $tr_{sol} : I+\{sol\} \rightarrow I+\{sol,sol_1,sol_2\}$ est une injection canonique ;
- $E_i[sol \rightarrow sol_i]$ est l'ensemble des (in)équations obtenues par remplacement syntaxique de chaque occurrence de sol par sol_i dans E_i.

L'index spécial *sol* étant identique dans β_1 et β_2, il est tout d'abord renommé en sol_1 et sol_2 respectivement. Ensuite, la somme de sol_1 et sol_2 donne la longueur totale des séquences concaténées.

OPÉRATIONS SUR LES SUBSTITUTIONS ABSTRAITES

Normalisation d'une substitution

CleanUp(β) = NORMALIZE(β) = β'.

Soient :
$\beta = \langle sv,mo,ty,frm,ps,E \rangle$ définie sur I ;
$dom(\beta) = \{X_1,...,X_n\}$;
$\beta' = \langle sv',mo',ty',frm',ps',E' \rangle$ définie sur I' ;
$\#I' \leq \#I$.

L'implantation est :

If $\exists\, i \in I \wedge k \in \{1,...,n\} : (sv(X_k) \preceq i \wedge i \preceq i)$ (càd $\exists$ circuit) then
    $\beta' = \bot$
Else
    I' = I
    while $I \neq \varnothing$ do
        let $i \in I$
        if $(\forall\, k \in \{1,...,n\} : \neg(sv(X_k) \preceq i)$ then
            $I' = I' / \{i\}$
        $I = I / \{i\}$
    tr : I'→I : $tr_{II'}$ est la fonction d'identité
    $sv' = sv_{II'}$
    $\langle mo',ty',frm',ps',E' \rangle = tr^{<}\langle mo,ty,frm,ps,E \rangle$

Opération Ext_lub

EXT_LUB(β_1,β_2) = $\langle \beta', tr_1, tr_2, st \rangle$.

Cette opération retourne $\langle \beta',tr_1,tr_2 \rangle$ = LUB(β_1,β_2). De plus, elle retourne st = vrai si β' est l'union stricte de β_1 et β_2 (-c.à.d.-$Cc(\beta')=Cc(\beta_1) \cup Cc(\beta_2)$).

L'union de deux substitutions abstraites

LUB(β_1,β_2) = (β,tr_1,tr_2) = UNION(β_1,β_2).

Soient :
$\beta_1 = \langle sv_1,mo_1,ty_1,frm_1,ps_1,E_1 \rangle$ définie sur $I_{p_1} = \{1,...,p_1\}$ une substitution abstraite ;
$\beta_2 = \langle sv_2,mo_2,ty_2,frm_2,ps_2,E_2 \rangle$ définie sur $I_{p_2} = \{1,...,p_2\}$ une substitution abstraite ;
$dom(\beta_1) = dom(\beta_2) = D = \{X_1,...,X_n\}$;
$\beta = \langle sv,mo,ty,frm,ps,E \rangle$ une substitution pseudo-abstraite définie sur $I_p=\{1,...,p\}$;
$tr_1 : I' \rightarrow I_1$ le mapping structurel de β' sur β_1 ;

tr_2 : $I' \rightarrow I_2$ le mapping structurel de β' sur β_2 ;
$dom(\beta) = D$.

L'implantation de cette opération se fait en plusieurs étapes :

1) On établit une correspondance entre les termes des deux substitutions abstraites β_1 et β_2. Cette correspondance est représentée par une fonction $t : F \rightarrow I_p$ dans laquelle F est un ensemble de paires d'index, $I_p=\{1,...,p\}$ et $p = \#F$.

Construction de F et t.

On commence par définir un ensemble de correspondances E basé sur les composants sv des substitutions :

$$E = \{(i,j) \mid \exists X \in D : i = sv_1(X) \wedge j = sv_2(X) \}.$$

Ensuite, on complète E avec les correspondances des termes qui composent d'autres termes :

$$F = E \cup \{(i_k,j_k) \mid 1 \leq k \leq n \wedge (i,j) \in E \wedge frm_1(i)=f(i_1,...,i_n) \wedge frm_2(j)=f(j_1,...,j_n)\}$$

Si les termes correspondants n'ont pas la même forme, on ne définit pas de forme dans le résultat. Ainsi, les termes générés par le résultats auront toutes les formes possibles, et donc, auront au moins une fois les formes des termes i et j.

On construit t sur base de F :

$$t : F \rightarrow I_p \ (p=\#F).$$

De plus, on construit les mappings $tr_1 : I_p \rightarrow I_{p_1}$ et $tr_2 : I_p \rightarrow I_{p_2}$ qui définissent les correspondances des index entre les substitutions β_1 et β et entre les substitutions β_2 et β.

$tr_1(t(i,j)) = i \ \forall \ (i,j) \in F$
$tr_2(t(i,j)) = j \ \forall (i,j) \in F$.

2) L'opération LUB est défini par :

$sv(X) = t(sv_1(X),sv_2(X)) \ \forall \ X \in D$
$\forall \ (i,j) \in F : \quad mo(t(i,j)) = LUB_{mode}(mo_1(i),mo_2(j))$
$\quad ty(t(i,j)) = LUB_{type}(ty_1(i),ty_2(j))$
$frm = \{\langle t(i,j),f(t(i_1,j_1),...,t(i_n,j_n))\rangle \mid (i,j) \in F \wedge frm_1(i)=f(i_1,...,i_n) \wedge frm_2(j)=f(j_1,...,j_n)\}$
$ps = \{(t(i,j),t(i',j')) \mid (i,j) \in F \wedge (i',j') \in F \wedge (ps_1^*(i,i') \vee ps_2^*(j,j')) \wedge$
$\quad frm(t(i,j))=frm(t(i',j'))=undef\}$
$E = tr_1^{<}(E_1) \cup tr_2^{<}(E_2)$

La comparaison de deux substitutions abstraites

$\beta_1 \leq \beta_2$.

Soient :
$\beta_1 = \langle sv_1,mo_1,ty_1,frm_1,ps_1,E_1 \rangle$ définie sur I_1 ;
$\beta_1 = \langle sv_2,mo_2,ty_2,frm_2,ps_2,E_2 \rangle$ définie sur I_2 ;
$dom(\beta_1) = dom(\beta_2) = \{X_1,...,X_n\}$.

Pour déterminer si $\beta_1 \leq \beta_2$, il est nécessaire de définir un mapping des index de β_1 sur les index de β_2 puisque le même index peut référencer des termes complètement différents dans les deux substitutions. Soit $t : I_2 \rightarrow I_1$ la fonction réalisant ce mapping défini par les conditions :
i) $\forall \ X \in D : sv_1(X)=t(sv_2(X))$;
ii) $\forall \ i,i_1,...,i_q \in I_2 : frm_2(i)=f(i_1,...,i_q) \Rightarrow frm_1(t(i))=f(t(i_1),...,t(i_q))$.

On a $\beta_1 \leq \beta_2$ ssi on a :
i) $\forall \ i \in I_2 : mo_1(t(i)) \leq mo_2(i)$;
ii) $\forall \ i,j \in I_2 : frm_2(i)=frm_2(j)=undef : ps_1^*(t(i),t(j)) \Rightarrow ps_2(i,j)$.

Ces conditions peuvent aussi être définies par $tr^{<}(\alpha_1) \leq \alpha_2$ (où $\alpha_k = \langle sv_k, mo_k, ty_k, frm_k, E_k \rangle$) -càd que chaque composant de α_1 est plus précis ou égal aux composants de α_2, la fonction tr établissant la correspondance appropriée entre les termes des deux substitutions.

« Jointure » de deux substitutions abstraites

$JOIN_s(\beta 1,\beta 2) = GLB_{subst}(\beta_1,\beta_2) = \langle \beta', tr_1, tr_2 \rangle$.

Soient :
$\beta_1 = \langle sv_1, mo_1, ty_1, frm_1, ps_1, E_1 \rangle$ définie sur I_1 ;
$\beta_2 = \langle sv_2, mo_2, ty_2, frm_2, ps_2, E_2 \rangle$ définie sur I_2 ;
$dom(\beta_1) \cup dom(\beta_2) = D'$;
$dom(\beta_1) \cap dom(\beta_2) = D$;
$\beta' = \langle sv', mo', ty', frm', ps', E' \rangle$ définie sur I' ;
$dom(\beta') = D'$;
tr_1 le mapping structurel de β_1 sur β' ;
tr_2 le mapping structurel de β_2 sur β'.

L'implantation est :

$\langle sv', frm', tr_1, tr_2 \rangle = JOIN_{sf}(\langle sv_1, frm_1 \rangle, \langle sv_2, frm_2 \rangle)$
$mo' = GLB_{mode}(tr_1^{>}(mo_1), tr_2^{>}(mo_2))$
$ty' = GLB_{type}(tr_1^{>}(ty_1), tr_2^{>}(ty_2))$
$ps' = tr_1^{>}(ps_1^*, frm') \cap tr_2^{>}(ps_2^*, frm') \cap (leaves(I'))^2$
$E' = GLB(tr_1^{>}(E_1), tr_2^{>}(E_2))$

Remarque : si $dom(\beta_1) = dom(\beta_2)$, cette opération correspond au GLB de deux substitutions.

Opération JOINsf

$JOIN_{sf}(\langle sv_1, frm_1 \rangle, \langle sv_2, frm_2 \rangle) = \langle sv', frm', tr_1, tr_2 \rangle$.

Soient :
$sv_k : D_k \rightarrow I_k$ des fonctions totales avec D_k des ensembles de variables (k=1,2) ;
$frm_k : I_k \rightarrow Frames_{I_k}$ (k=1,2) ;
$D = D_1 \cup D_2$;
$in_k : D \rightarrow D_k$ (k=1,2) des injections ($\forall X \in D_k$, $in_k(X)=X$, $\forall X \in D/D_k$, $in_k(X)$ est indéfini ;
$tr_k : I_k \rightarrow I'$ des fonctions partielles ;
$sv' : D \rightarrow I'$ une fonction totale ;
$frm' : I' \rightarrow Frames_{I_k}$.

On a :

$in_1 : D \rightarrow D_1$, $sv_1 : D_1 \rightarrow I_1$, $tr_1 : I_1 \rightarrow I'$
$in_2 : D \rightarrow D_2$, $sv_2 : D_2 \rightarrow I_2$, $tr_2 : I_2 \rightarrow I'$

Opération Ref_{frm}

$Ref_{Frm}(\beta_1,\beta_2,tr_{1,2}) = \langle \beta', tr' \rangle$.

Soient :
$\beta_1 = \langle sv_1, mo_1, ty_1, frm_1, ps_1, E_1 \rangle$ définie sur I_1 ;
$\beta_2 = \langle sv_2, mo_2, ty_2, frm_2, ps_2, E_2 \rangle$ définie sur I_2 ;
$tr_{1,2} : I_1 \rightarrow I_2$ le structural mapping de β_1 sur β_2 ;
$\beta' = \langle sv', mo', ty', frm', ps', E' \rangle$ définie sur I' ;
$tr' : I_1 \rightarrow I_2$.

L'implantation est :

$\beta^0 = \beta_1$
$tr^0 = tr_{1,2}$
On calcule la séquence de substitutions abstraites $\beta^0,...,\beta^i,...$ et les mappings structurels $tr^0,...,tr^i,...$
Soit β^i et tr^i : $I^i \rightarrow I_2$ la substitution et le mapping courants, on suppose qu'il existe un $j \in I^i$ tel que $mo^i(j) \leq novar$ et $frm^i(j)=undef$ et $frm_2(tr^i(j)) = f(k_1,...,k_n)$. Alors on calcule β^{i+1} et tr^{i+1} :

$I^{i+1} = I^i \cup \{j_1,...,j_n\}$ où $j_1,...,j_n$ sont des nouveaux indices distincts
$sv^{i+1} = sv^i$
$frm^{i+1} = frm^i \cup \{j \rightarrow f(j_1,...,j_n)\}$
$tr^{i+1} = tr^i \cup \{j_1 \rightarrow k_1,...,j_n \rightarrow k_n\}$
$mo^{i+1}(j) = mo^i(j)\ \forall\ j \in I^i$
$(mo^{i+1}(j_1),...,mo^{i+1}(j_n)) = EXTR_{mode}(mo^i(j),f)$
$ty^{i+1}(j) = ty^i(j)\ \forall\ j \in I^i$
$(ty^{i+1}(j_1),...,ty^{i+1}(j_n)) = EXTR_{type}(ty^i(j),f,mo^i(j),mo^{i+1}(j_1),...,mo^{i+1}(j_n))$
$ps^{i+1} = ps^i \cup \{(j_l,k) \mid l \in \{1,...,n\} \wedge mo^{i+1}(j_l) \neq ground \wedge (j,k) \in ps^i\}$

Sinon $\beta' = \beta^i$ et $tr' = tr^i$.

Opération Ref_{α}

$Ref_{\alpha}(\beta_1, \beta_2, tr_{1,2}) = (\beta', tr')$.

Soient :
$\beta_1 = \langle sv_1, mo_1, ty_1, frm_1, ps_1, E_1 \rangle$ définie sur I_1 ;
$\beta_2 = \langle sv_2, mo_2, ty_2, frm_2, ps_2, E_2 \rangle$ définie sur I_2 ;
$tr_{1,2} : I_1 \rightarrow I_2$ le structural mapping de β_1 sur β_2 ;
$\beta' = \langle sv', mo', ty', frm', ps', E' \rangle$ définie sur I' ;
$tr' : I_1 \rightarrow I_2$.

L'implantation est :

$I' = I_1$
$sv' = sv_1$
$mo'(i) = GLB_{mode}(mo_1(i), mo_2(tr_{1,2}(i)))\ \forall\ i \in I'$
$ty'(i) = GLB_{type}(ty_1(i), ty_2(tr_{1,2}(i)))\ \forall\ i \in I'$
$ps' = ps_1$
$tr' = tr_{1,2}$

Opération Ref_{Ref}

$Ref_{Ref}(\beta_1, \beta_2, tr_{1,2}) = (\beta', tr')$.

Soient :
β_1 et β_2 deux substitutions abstraites définies sur I_1 et I_2 ;
$dom(\beta_1) = dom(\beta_2)$;
$tr_{1,2} : I_1 \rightarrow I_2$ le structural mapping entre β_1 et β_2 ;
β' une substitution abstraite définie sur I' ;
$tr' : I' \rightarrow I_2$ le structural mapping de β' sur β_2 ;
$dom(\beta') = dom(\beta_1) = dom(\beta_2)$.

Implantation :

$\langle \beta_3, tr_{3,2} \rangle = Ref_{Frm}(\beta_1, \beta_2, tr_{1,2})$
$\langle \beta', tr' \rangle = Ref_{\alpha}(\beta_3, \beta_2, tr_{3,2})$

OPÉRATIONS SUR LES SÉQUENCES ABSTRAITES

Préordre sur les séquences abstraites

$B_1 \leq B_2$.

Soient :
$B_k = \langle \beta_{in}, \beta_{ref,k}, \beta_{out,k}, E_{ref\_out,k}, Es_{ol,k} \rangle$ (k = 1,2).

Les deux séquences ont la même substitution abstraite en entrée.

Le préordre est défini par :

$$B_1 \leq B_2 \Leftrightarrow \quad \beta_{ref,1} \leq \beta_{ref,2} \quad (\exists\, tr_{ref} : I_{ref,2} \rightarrow I_{ref,1})$$
$$\beta_{out,1} \leq \beta_{out,2} \quad (\exists\, tr_{out} : I_{out,2}, I_{out,1})$$
$$(tr_{ref} + tr_{out})^{<}(E_{ref\_out,1}) \leq E_{ref\_out,2}$$
$$(tr_{ref} + \{sol \rightarrow sol\})^{<}(E_{sol,1}) \leq E_{sol,2}$$

Normalisation d'une séquence abstraite

NORMALIZE(B) = B'.

Soient :
$B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$;
$B' = \langle \beta_{in}', \beta_{ref}', \beta_{out}', E_{ref\_out}', E_{sol}' \rangle$.

L'implantation est :

$\beta_{in}' = \beta_{in}$
$\beta_{ref}' = JOINs(\beta_{in}, \beta_{ref})$
$\beta_{out}' = Ref_{out}(\beta_{ref}', \beta_{out})$
$E_{ref\_out}' = Ref_{ref\_out}(\beta_{ref}, \beta_{ref}', E_{out}, \beta_{out}, \beta_{out}')$
$E_{sol}' = Ref_{sol}(\beta_{ref}, \beta_{ref}', E_{sol})$.

Création d'une séquence abstraite initiale

Empty_SA(β) = B.

$\forall\, \theta \in Cc(\beta) : <\theta,<>> \in Cc(B)$.

L'implantation est :

$B = <\beta, \perp, \perp, \perp, \{sol = 0\}>$.

Concaténation de séquences abstraites

CONC_SA(B_1,B_2) = B'.

Soient :
$B_k = \langle \beta_{in}, \beta_{ref,k}, \beta_{out,k}, E_{ref\_out,k}, E_{sol,k} \rangle$ (k=1,2) deux séquences abstraites ;
$dom_{out}(B_1) = dom_{out}(B_2)$;
$B' = \langle \beta_{in}', \beta_{ref}', \beta_{out}', E_{ref\_out}', E_{sol}' \rangle$ une séquence pseudo-abstraite ;
$dom_{in}(B') = dom_{in}(B_k)$ (k=1,2) ;
$dom_{out}(B') = dom_{out}(B_k)$ (k=1,2).

L'implantation est :

$\beta_{in}' = \beta_{in}$
$\langle \beta_{ref}', tr_{ref,1}, tr_{ref,2}, st \rangle = EXT\_LUB(\beta_{ref,1}, \beta_{ref,2})$

$\langle \beta_{out}', tr_{out,1}, tr_{out,2} \rangle = LUB(\beta_{out,1}, \beta_{out,2})$
$E_{ref\_out}' = LUB((tr_{out,1} + tr_{ref,1})^{<}(E_{ref\_out,1}), (tr_{out,2} + tr_{ref,2})^{<}(E_{ref\_out,2})$
$\langle \beta_{int}, tr_{int,1}, tr_{int,2} \rangle = JOIN_s(\beta_{ref,1}, \beta_{ref,2})$
$E_{ref,sol,1} = (tr_{int,1} + \{sol \rightarrow sol\})^{>}(E_{sol,1})$
$E_{ref,sol,2} = (tr_{int,2} + \{sol \rightarrow sol\})^{>}(E_{sol,2})$
tr_{sol} : $\{sol\} \rightarrow I_{ref}'$ + {sol} est une injection canonique
si st alors
$E_{sol}' = LUB((tr_{ref,1} + \{sol \rightarrow sol\})^{<}(E_{sol,1}),$
$(tr_{ref,2} + \{sol \rightarrow sol\})^{<}(E_{sol,2}),$
$(tr_{inter} + \{sol \rightarrow sol\})^{<}(SUM_{sol}(E_{ref,sol,1}, E_{ref,sol,2})))$
sinon
$E_{sol}' = LUB((tr_{ref,1} + \{sol \rightarrow sol\})^{<}(E_{sol,1}),$
$(tr_{ref,2} + \{sol \rightarrow sol\})^{<}(E_{sol,2}),$
$(tr_{inter} + \{sol \rightarrow sol\})^{<}(SUM_{sol}(E_{ref,sol,1}, E_{ref,sol,2})),$
$tr^{>}_{sol}([[sol=0]]))$

On voit que la concaténation du nombre de solutions des deux séquences ne correspond pas à la borne supérieure du nombre de solutions de la première et de la deuxième séquence, mais correspond à la somme du nombre de solutions de chaque séquence.

Pour bien évaluer E_{sol}', il est nécessaire de détecter les clauses mutuellement exclusives. Pour cela, on calcule d'abord le glb des composants β_{ref} des deux séquences (c'est à dire, l'ensemble des substitutions abstraites qui réussissent au moins une fois dans les deux séquences). Si le glb est $\perp$, les clauses sont mutuellement exclusives, aucune substitution ne réussissant dans les deux clauses. On ne retient que le nombre de solutions des deux séquences. Si le glb n'est pas $\perp$, E_{sol}' est égal au nombre de solutions du glb.

Intersection de deux séquences abstraites

JOIN_SA(B_1,B_2) = B'.

Soient :
$B_k = \langle \beta_{in,k}, \beta_{ref,k}, \beta_{out,k}, E_{ref\_out,k}, E_{sol,k} \rangle (k = 1,2)$;
$dom_{in}(B_2) \subseteq dom_{in}(B_1)$;
$dom_{out}(B_2) \subseteq dom_{out}(B_1)$;
$B' = \langle \beta_{in}', \beta_{ref}', \beta_{out}', E_{ref\_out}', E_{sol}' \rangle$;
$dom_{in}(B') = dom_{in}(B_1)$;
$dom_{out}(B') = dom_{out}(B_1)$.

L'implantation est :

$\langle \beta_{in}', tr_{in,1}, tr_{in,2} \rangle = JOIN_s(\beta_{in,1}, \beta_{in,2})$
$\langle \beta_{ref}', tr_{ref,1}, tr_{ref,2} \rangle = JOIN_s(\beta_{ref,1}, \beta_{ref,2})$
$\langle \beta_{out}', tr_{out,1}, tr_{out,2} \rangle = JOIN_s(\beta_{out,1}, \beta_{out,2})$
$E_{out}' = GLB_E((tr_{ref,1} + tr_{out,1})^{>}(E_{out,1}), (tr_{ref,2} + tr_{out,2})^{>}(E_{sol,2}))$
$E_{sol}' = (GLB_E((tr_{ref,1} + \{sol \rightarrow sol\})^{>}(E_{sol,1}), (tr_{ref,2} + \{sol \rightarrow sol\})^{>}(E_{sol,2}))$.